USENIX BALTIMORE CONFERENCE PROCEEDINGS SUMMER 1989

U
US
USE
U NIX
USENIX

CONFERENCE PROCEEDINGS

Baltimore, Maryland
June 12 - 16, 1989

USENIX Association

Proceedings of the Summer 1989 USENIX Conference

June 12 — June 16, 1989
Baltimore, Maryland USA

For additional copies of these proceedings contact
USENIX Association
2560 Ninth St., Suite 215
Berkeley, CA 94710 USA

The price is $20.
Outside the U.S. and Canada please add
$15 per copy for postage (via air printed matter)

TABLE OF CONTENTS

PROGRAM DEVELOPMENT

Thursday (11:00-12:30) Chair: Steven Bellovin

FILE SYSTEMS

Thursday (2:00-3:00) Chair: Armando Stettner

WORK IN PROGRESS

Thursday (3:30-5:30) Chair: Peter Honeyman

PREFACE

Welcome to the Summer 1989 USENIX Technical Conference and Exhibition. The Program Committee would like to thank all of the contributors and believes that we have an interesting collection of papers for presentation. As has been the case for the last two conferences, full papers were required for consideration and peer review. Our review and production task began immediately after the Winter conference in San Diego so we have included a Work-In-Progress on Thursday afternoon to showcase more recent developments.

While assembling our agenda we decided to invite three guest speakers to Baltimore to give us all a broader outlook on computing. Bill Wulf has led several pioneering efforts, Rob Pike introduced many of us to "windows", and Tom Malarkey has set guidelines for privacy and integrity of data. We hope you enjoy our selection.

When I was first introduced to UNIX, I was told that learning about the system would be a bit like climbing a chimney while inside it. That is, there is no clear process but you must climb all four walls at once and press outwards in order to travel upwards. The path is not linear and all sorts of things will rub off on you while you learn. I soon found myself with documentation and source code all around me and I learned editors, programming languages, an operating system, and a bit about hardware all at the same time. In the years since then many have mastered UNIX and extended it; we can now stand on their shoulders and reach ever higher.

I would like to thank the USENIX organization for providing us with this conference forum and all of you for attending and participating. Beyond the program committee I thank Judy DesHarnais for planning assistance and Evi Nemeth and her staff for publishing these proceedings. In addition, I would like to thank Sun Microsystems for allowing me time and resources to perform my duties as program chair.

Neil Groundwater
Program Chair

CONFERENCE COMMITTEE

CONFERENCE ORGANIZERS

Neil Groundwater, *Technical Program Chair*
(Sun Microsystems, Inc.)

Judith F. Desharnais, *Meeting Planner*,
(USENIX Association)

John Donnelly, *Tutorial Coordinator*
(USENIX Association)

Evi Nemeth, *Proceedings Production*
(University of Colorado, Boulder)

John Donnelly, *Vendor Exhibition Manager*
(USENIX Association)

TECHNICAL PROGRAM COMMITTEE

Steven Bellovin
(AT&T Bell Laboratories)

Matthew Hecht
(IBM Corporation)

Peter Honeyman
(CITI/University of Michigan)

Dan Klein
(Software Engineering Institute, CMU)

Matt Koehler
(Sun Microsystems, Inc.)

Don Libes
(National Inst. of Standards and Technology)

Harold Melanson
(Cray Research)

Armando Stettner
(Digital Equipment Corp.)

Pat Wilson
(University of Virginia)

Jean Wood
(MIPS Computer Systems, Inc.)

TECHNICAL PROGRAM REVIEWERS

Geoff Arnold
Jim Barrett
James M. Bodwin
Harry Bovik
Heather Burris
Scott Callon
Richard Campbell
Tom Cargill
Anne Chenette
Alex Colvin
Robert J. Cordero
Elizabeth J. Cummings
Nelson Fernandez
Ken Fisher
Coni Gehler
Ron Gomes
Andrew Grimshaw
Lori Grob
Cathy Guetzlaff
Ted Hanss
Mark Himelstein
Anne M. Holler
Byron Howes
Andrew Hume
Percy Irani
Edith C. Irwin
James A. Jokl
Bryan Koch
Andy Koenig
Dave Korn
Tom Kramer
Eduardo Krell
David Kucharczyk
Samuel J. Leffler
Steve Losen
Ken Manheimer
Doug McIlroy
M. Kirk McKusick
David Miller
Nancy Mintz
Barbara Moo
Mike Padovano
Andrew S. Partan
Randy Pausch
Dave Presotto
Carol Preston
Charlie Price
John Puttress
Bill Sherman
Tim Sigmon
Velu Sinha
Bruce Smith
Griff Smith
Mark Smith
Michael T. Stolarchuk
Tim Strayer
Susan Symington
Judy Szikora
Phong Vo
Ed Whelan
Andrew Wild

PROCEEDINGS PRODUCTION

Evi Nemeth, Dotty Foerst, Trent Hein,
Paul Kooros, Laszlo Nemeth, Brett Reid,
Steven Roth, and Tyler Stevens
(University of Colorado)

AUTHOR INDEX

A Watermark-based Lazy Buddy System for Kernel Memory Allocation

T. Paul Lee
AT&T Bell Laboratories
Holmdel, NJ 07733

R. E. Barkley
AT&T Bell Laboratories
Summit, NJ 07901

ABSTRACT

In this paper we describe a watermark-based lazy control policy of the buddy system for managing UNIX® kernel memory space. This policy achieves low operational costs by capitalizing on the steady-state behavior of memory demands. The lazy control policy maintains watermark indices to track the use of buffers for each size. Depending on these indices, the policy can be in *lazy* mode, *reclaiming* mode, or *accelerated* mode when dealing with buffer coalescing. We incorporate a damping mechanism in the coalescing process by anticipating that the buffer will soon be re-used, and an acceleration mechanism that lets the policy detect and react to shifting memory demands. The controls to these mechanisms are distributed over each class of buffer size; such distributed controls are particularly attractive when the costs of maintaining global information and control are prohibitive.

We present results and comparisons of experiments conducted on a prototype lazy buddy system used to manage UNIX System STREAMS buffers. The costs of buffer operations under the lazy control policy are consistently lower than those under the standard buddy system with only a slight increase in blocking probability.

1. INTRODUCTION

The buddy system[1] [2] is a dynamic memory manager that responds well to shifting memory demands[1]. The costs to allocate or free a buffer are fairly inexpensive compared to those of best- or first-fit policies. However, these costs are not insignificant when compared to those of a memory system with a static configuration. Studies of UNIX® System STREAMS buffer request/release activities[3] in UNIX System

1. A binary buddy system is assumed here for ease of discussion; the lazy control of the coalescing process can be applied to Fibonacci and weighted buddy systems.

V show that workloads often exhibit steady-state behavior in memory demand; that is, they maintain slow time-varying demand for buffers of particular sizes. (We refer to buffers of the same size as a buffer *class*.) We have designed a *lazy* control policy for the buddy system that capitalizes on this steady-state behavior to reduce the costs of buffer operations.

The policy maintains indices that measure how heavily each buffer class is being used. When the index for a particular class shows steady usage (i.e. the index remains below a low watermark), buffer releases are only local; that is, a released buffer is not marked as free in the system-wide bit-map but is placed on the free list for the class. This heuristic eliminates expensive bookkeeping and avoids the cost of coalescing memory. Since buffer demands are steady, the buffer is likely to be reallocated; the system thus avoids the cost of splitting memory and eliminates additional bookkeeping. When the index for a buffer class rises above the low watermark, the system marks a released buffer as free in the bit-map and coalesces the buffer with its buddy if the buddy is free. If the index shows the class is under-utilized (i.e. the index crosses a high watermark), the system doubles the rate of coalescing by releasing and coalescing one additional locally freed buffer. In other words, each buffer class keeps a dynamic cache of buffers; the size of the cache depends on the current demand for buffers of that size. Furthermore, each buffer class meters itself to detect and react to shifting memory demands.

In essence, we have added a damping mechanism to the buddy system's coalescing process by anticipating that the buffer will soon be reused. That is, we assume temporal locality in buffer reusage behavior. We have also added an acceleration mechanism to speed up the buffer coalescing process to respond to shifting memory demands. The controls to these mechanisms are distributed over each class of buffer size; such distributed controls are especially attractive when the costs of maintaining global information and control are prohibitive.

The body of this paper is organized as follows: in Section 2 we describe the lazy control policy and its characteristics. In Section 3, we present measurements made on a prototype UNIX System STREAMS buffer manager based on the lazy control policy. Finally, we summarize the observations and results in Section 4.

2. THE LAZY CONTROL POLICY(θ_l,θ_s)

To set the framework for discussing the lazy control policy, we first briefly describe the static system and the standard buddy system. In the static system, the number of buffers in each class is predetermined. Initially, all buffers of a particular size are linked together in a free list. When a buffer is requested, the memory manager simply takes a buffer from the front of the list; when the buffer is returned, it is relinked into the list. Thus, the costs to allocate or free a buffer are minimal. Unfortunately, predetermining the correct buffer configuration is a difficult task[3]. If we configure too few buffers for a particular class, buffer requests will fail; if we configure too many, we waste memory. Furthermore, a configuration optimal for one workload may not be adequate for another. Static memory allocation schemes, although inexpensive, are typically characterized by high failure rates or high *blocking probability.*

The buddy system addresses both of these problems. It begins with a monolithic piece of memory. In response to a buffer request, the system splits the free memory into smaller pieces until it obtains a buffer that matches the request. In a binary buddy system a request for a buffer of size s is satisfied with the smallest buffer of size 2^n such that $s \leq 2^n$. The unused or free *buddy* of the allocated buffer is placed on the free list corresponding to its class. Since this buffer is free, buffer descriptors such as buffer size and forward and backward pointers are kept in the buffer itself[2]. The system responds to additional requests for buffers either by taking a properly sized buffer from its free list or by taking a larger buffer and splitting it until a correctly sized buffer is obtained. The system keeps the state of every buffer, whether allocated or free, in a system-wide bit-map. When a buffer is freed, the system looks in the bit-map to see if the buddy of the buffer is also free. If it is, the system coalesces the memory to make a larger free buffer; the larger buffer is then coalesced with its buddy if that buddy is free. Since the system recursively coalesces whenever any buffer is freed, when all buffers have been freed the system memory returns to its original monolithic state. Because it coalesces memory whenever possible and is not confined to a predetermined buffer configuration, the buddy system has a much lower blocking probability than the static system. However, it is more expensive than the static system because of the bookkeeping costs and the time spent splitting and coalescing memory.

The watermark-based lazy control policy of the buddy system is designed to have the low cost characteristics of the static allocation policy and the low blocking probability characteristics of the buddy system. The lazy buddy system differs from the buddy system in that it modulates the buddy coalescing process depending on the mode of each buffer class.

2.1 A Self-Metering and Self-Policing Mechanism

In the lazy buddy system, each buffer class can be in one of three modes: *lazy*, *reclaiming*, or *accelerated*. The mode of a class is determined by the value of the buffer idle index, θ, relative to two watermark threshold parameters, θ_l, the *lazy ratio*, and θ_s, the *speedup ratio*. The three modes are defined as follows:

$$\begin{array}{rl} \text{lazy:} & \text{if } \theta \leq \theta_l \\ \text{reclaiming:} & \text{if } \theta_l < \theta \leq \theta_s \\ \text{accelerated:} & \text{if } \theta_s < \theta \end{array}$$

To calculate θ, the system maintains two counters for each buffer class; one is the total number of buffers of that size currently existing (N), and the other is the total number of buffers in the free list (F), i.e., buffers neither allocated nor coalesced. The ratio of these two counters is the buffer idle index ($\theta = F/N$ for $N > 0$). Intuitively, the index reflects the state of buffer utilization. If the buffer class is well-utilized, a buffer release is a local event and the system does not try to coalesce. If buffer class is under-utilized, a buffer release triggers the coalescing process. If the index goes beyond the speedup threshold, the rate of buffer coalescing process doubles; by doubling the coalescing rate, we ensure that all buffers will be coalesced if all are returned.

To distinguish buffers freed in the global bit-map from those freed only locally, each buffer in the free list has a binary state variable indicating whether the buffer is *delayed* or *not-delayed*. A "delayed" buffer is only free in the free list and it can only be accessed through the corresponding list head; the system-wide bit-map shows the buffer as still allocated. For "not-delayed" buffers, the bit-map has the buffers marked free.

2. Assuming pointers are 4 bytes long, this limits the size of the smallest buffer maintained by most buddy systems to 16 bytes.

The self-policing mechanism works as follows: when a buffer is returned to the system, the free counter for the corresponding buffer class is incremented first and the current mode of the class is determined. If the class is in *lazy* mode, the freed buffer is marked "delayed" and is linked to the list without any further processing. If the class is in *reclaiming* mode, the freed buffer is marked "not-delayed" and is freed according to the usual buddy system coalescing rules. If the class is in *accelerated* mode, the buffer is freed as in the reclaiming mode, and an additional "delayed" buffer from the list (if any) is marked "not-delayed" and is freed by the usual coalescing rules. Thus, the worst case for a free operation is bounded because we release at most one extra locally free "delayed" buffer. Buffer coalesces are recursive; a newly coalesced buffer from one class becomes a buffer release for the next larger class, and this buffer is freed according to the lazy control policy for its class.

```
Given in state $S(N,F_d,F_n)$, the next state $S_{next}$ after an operation is as follows:

(I). if the next operation is a buffer request,
    If $F_d>0$, then $S_{next}$ becomes $(N,F_d-1,F_n)$,
    Else if $F_n>0$, then $S_{next}$ becomes $(N,0,F_n-1)$.
    When $F_d=F_n=0$, first get two buffers by splitting a larger one, and
        then $S_{next}$ becomes $(N+2,1,0)$ if $\theta_l(N+2)\geq 1$ and $(N+2,0,1)$ otherwise.

(II). if the next operation is a buffer free,
    Let $F'=F_d+F_n+1$
    If $\theta_l N\geq F'$, then
      $S_{next}$ becomes $(N,F_d+1,F_n)$.
    Else if $\theta_s N\geq F'>\theta_l N$, then
      when $F_n>0$,
          $S_{next}$ becomes $(N,F_d,F_n+1)$ without any coalescing.
          $S_{next}$ becomes $(N-2,F_d,F_n-1)$ with one instance of coalescing.
      when $F_n=0$, $S_{next}$ becomes $(N,F_d,1)$.
    Else if $F'>\theta_s N$, then
      when $F_d=0$,
          $S_{next}$ becomes $(N,0,F_n+1)$ without any coalescing.
          $S_{next}$ becomes $(N-2,0,F_n-1)$ with one instance of coalescing.
      when $F_d>0$,
          $S_{next}$ becomes $(N,F_d-1,F_n+2)$ without any coalescing.
          $S_{next}$ becomes $(N-2,F_d-1,F_n)$ with one instance of coalescing.
          $S_{next}$ becomes $(N-4,F_d-1,F_n-2)$ with two instances of coalescing.
```

Figure 1. The Lazy Control Policy (θ_l,θ_s)

2.2 An Efficient One-List Implementation

We can keep both the "delayed" and "not-delayed" buffers on the same doubly-linked list by inserting freed buffers at the head or at the tail of the list according to their state. A "delayed" free buffer is inserted at the head of the list; a "not-delayed" free buffer (if its buddy is not free) is inserted at the tail of the list. Note that we always allocate buffers from the head of the list because those are the "delayed" buffers and are least expensive to allocate. With these rules and with an initially empty list, it is straightforward to show that the list is always composed of two distinct parts (with one or both possibly empty). This strategy makes the accelerated phase simple to implement; to reclaim an extra "delayed" buffer, we merely take the buffer from the head of the list. If the head of the list is not a "delayed" buffer, there are no "delayed" buffers in the list.

2.3 All Buffers Coalesced When All Returned

An important property of any dynamic memory management policy is that all buffers are coalesced when they are all returned. Obviously the buddy system has this property since it laboriously coalesces whenever possible. For the lazy version, we have to verify that, given the threshold parameter values, this property holds so that no buffer space becomes inaccessible.

To show that all buffers are coalesced when all are returned, we first define more precisely the lazy control policy(θ_l,θ_s). The state S of a class of buffers can be described by a non-negative triplet (N,F_d,F_n), where N is the total number of buffers of that class in existence, F_d is the number of "delayed" buffers in the free list, and F_n is the number of "not-delayed" buffers in the free list. The lazy control policy (θ_l,θ_s) is shown completely in Figure 1.

The state S of each buffer class is confined in the so-called *lazy space* through the use of two threshold parameters in the lazy coalescing control. This closure property enables us to coalesce back to the original memory state when all buffers are returned, and to bound the cost of freeing operation.

The **lazy space** L_N for integers $N\geq 0$ is defined to be the set of states (N,F_d,F_n) satisfying the following two conditions:

$$\text{(i).}\quad 0\leq F_n\leq \lceil N/2 \rceil,$$
$$\text{(ii).}\quad F_n\leq N-2F_d.$$

Condition (i) is a necessary condition of the coalescing rules for the globally freed buffers. Condition (ii) reflects the effect of boundedness (coalescing delays) on the number of "delayed" buffers. An example of the state transition diagram (for $N=6$) under the lazy control policy(¼,½) is depicted in Figure 2. To simplify the diagram, we do not show transitions for states with different Ns.

Given this framework, it is possible to show that all buffers are coalesced when all are returned, that is, $N=F_d+F_n$ implies $N=F_d=F_n=0$. The formal treatment can be found in[4]. Notice that if $\theta_l=0$, we have the vanilla buddy system, and if $\theta_l=\theta_s$, we have a simple two-mode (lazy and accelerated) policy. Intuitively, θ_l controls the extent of buffer under-utilization and θ_s controls the speed of response to shifting buffer demand.

3. A PROTOTYPE STUDY ON UNIX® SYSTEM V RELEASE 3.2

The STREAMS mechanism in UNIX® System V provides a flexible and unified approach to writing drivers and protocols[5] [6]. As part of its services, STREAMS supplies data buffers ranging in size from 4 to 4096 bytes in powers of 2; the number of buffers of each size is configured *a priori*. Static memory managers such as that used by STREAMS only work well if we can anticipate the buffer usage pattern and if that pattern does not change over time[3]. Because of these shortcomings, a buddy system has been implemented for consideration as the STREAMS buffer manager. To gather empirical performance data for the lazy control policy, we have implemented a lazy buddy prototype STREAMS buffer manager based on the original buddy system.

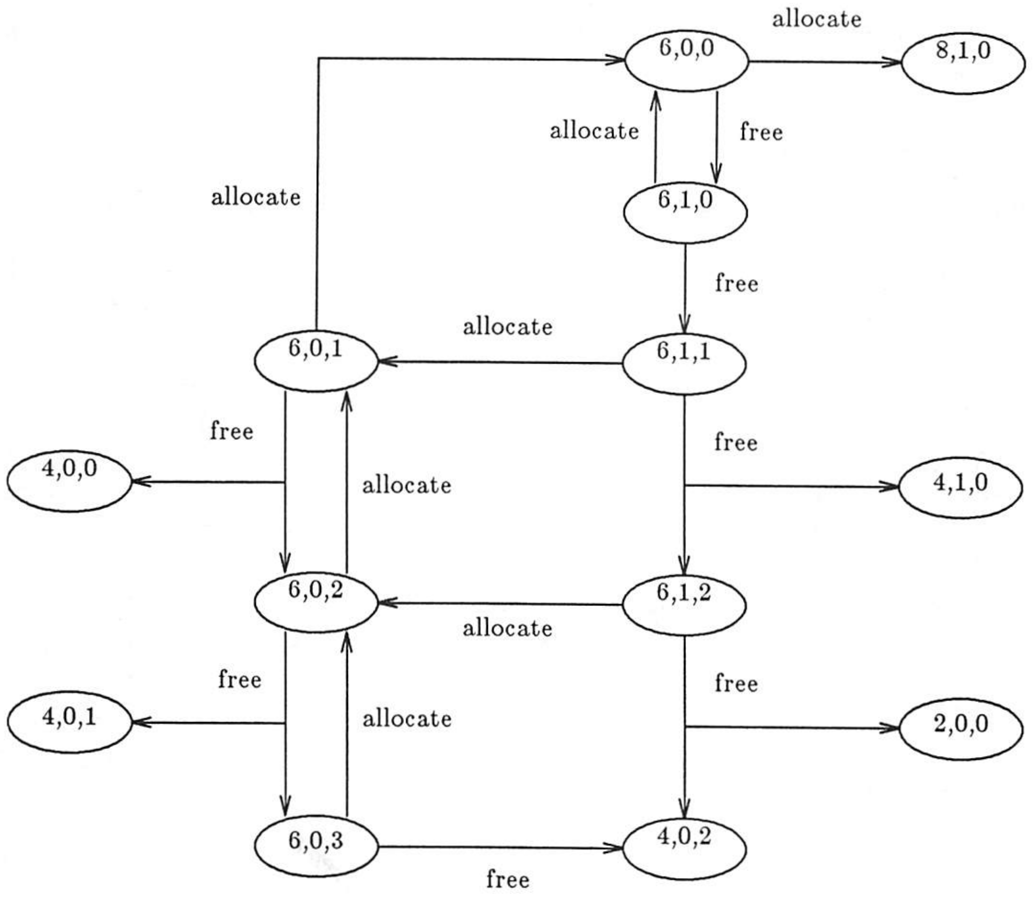


Figure 2. State Transition Diagram for Lazy Space $N=6$ Under (¼, ½) Policy

Although it is inexpensive to update counters and do mode testing, this accounting adds overhead not experienced by the standard buddy system. The lazy buddy system will only be useful if the savings from not updating global information and not splitting or coalescing memory more than offset the costs of bookkeeping and mode detection. To confirm that the lazy buddy system is indeed less expensive, we drove the prototype with STREAMS buffer traffic simulating a typical workload, measured the costs to allocate and free buffers in the lazy buddy system, and compared them with the costs measured on the standard buddy system. To simulate buffer traffic, we use a synthetic memory workload driver that schedules calls to the STREAMS functions that allocate a buffer, *allocb*(), and free a buffer, *freeb()*.

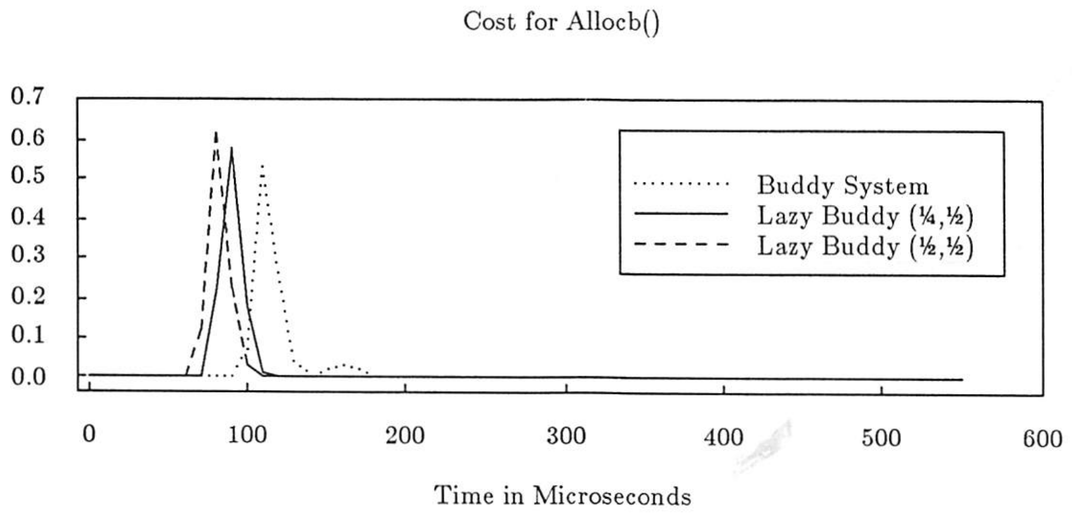
Although the costs of the buffer operations are important, we also need to consider the blocking probability or failure rate. To calculate this statistic, we count the total requests made and the number of requests that failed. For further comparison, we also provide the costs and blocking probability for the static memory manager.

3.1 Experimental Design for Performance Analysis

3.1.1 Synthetic Memory Workload Driver

We drive the memory system with a simulated multi-user timesharing workload[7]. Buffer demand traffic is characterized by a vector of 2-tuples that give the mean time between requests and mean buffer holding times for each class of buffer. We denote the average interarrival time for STREAMS buffer class i by $1/\lambda_i$ and denote the average holding time by $1/\mu_i$.

Given these two parameters, we can simulate activity for each STREAMS class in the following way: to begin, we schedule a request for a buffer of each class i at an average time $1/\lambda_i$. When it is time for the request, we call *allocb*() to allocate a buffer from class i, and then schedule the release of the buffer for an average time $1/\mu_i$ in the future. We also schedule the next request for a buffer in class i for an average time $1/\lambda_i$ in the future. When the release occurs, we free the STREAMS block via *freeb*(). As we make the calls to *allocb*() and *freeb*(), we time the routines by taking timestamps before we call them and timestamps after they return, and using the difference between the timestamps as the cost of the subroutine.



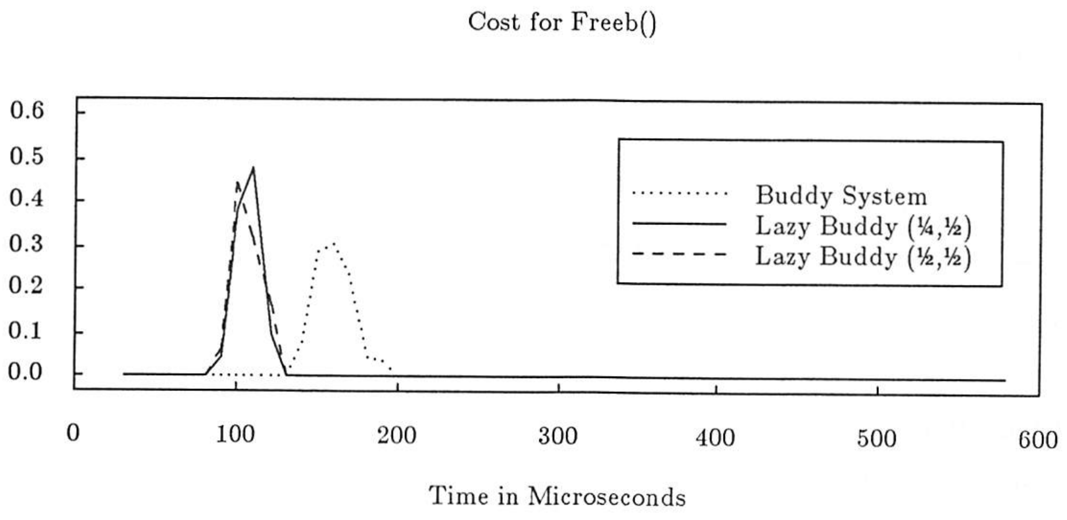
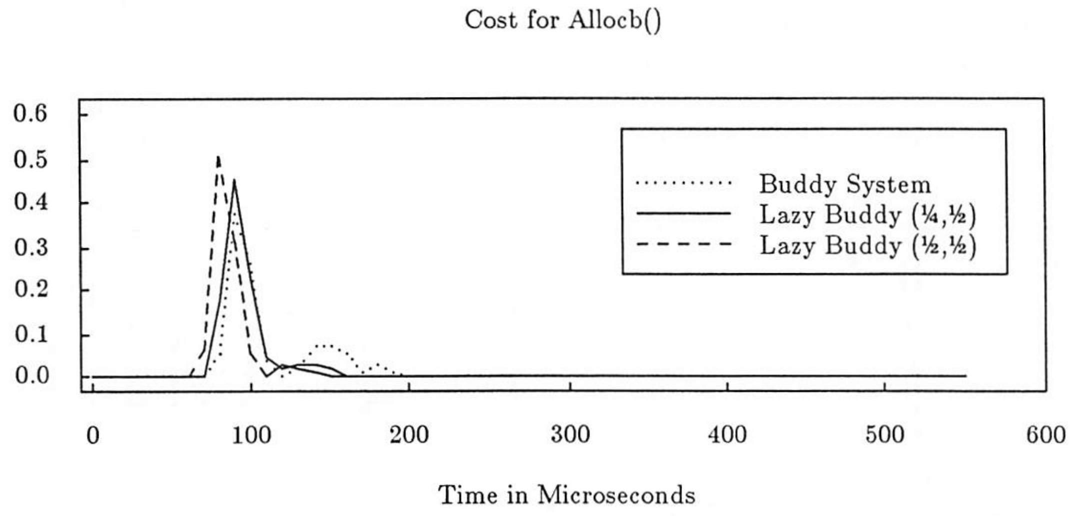


Figure 3. Probability Density Functions for Deterministic Interarrival and Service Times

Scheduling is done using the callout table. We use the system callout table to call an efficient callout mechanism based on a heap[8]. Every clock tick, the system callout mechanism invokes the local callout mechanism; the local callout mechanism calls *allocb*() and *freeb*() for all requests and releases that are scheduled for the current time, and then arranges for itself to be called again at the next clock tick.

The calls to *allocb*() and *freeb*() are timed using the high resolution timer supplied by the CASPER time trace tool[9]. CASPER also provides the capability to keep a detailed trace of each buffer request and release if requested. The detailed traces can be postprocessed to derive additional statistics and the distributions of times taken by *allocb*() and *freeb*().



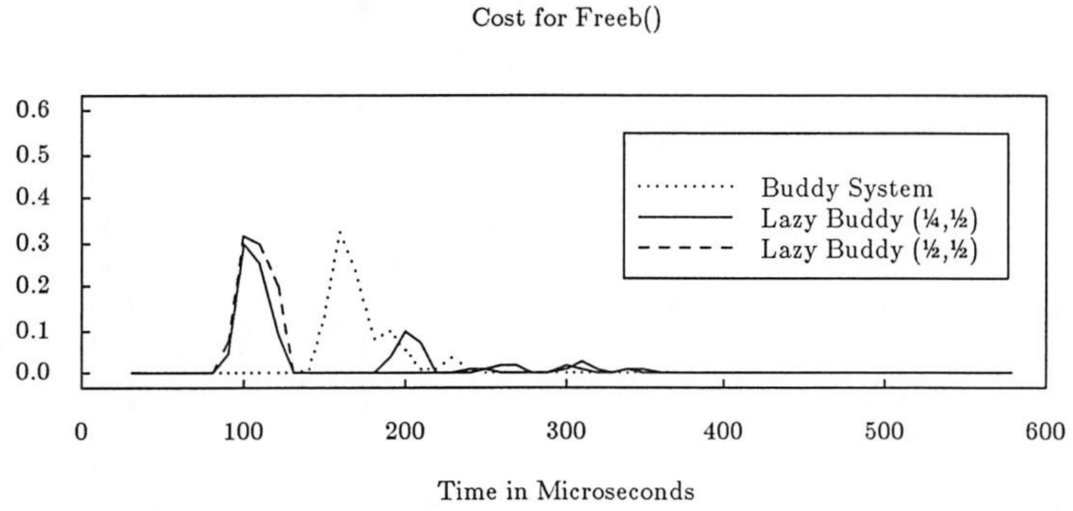
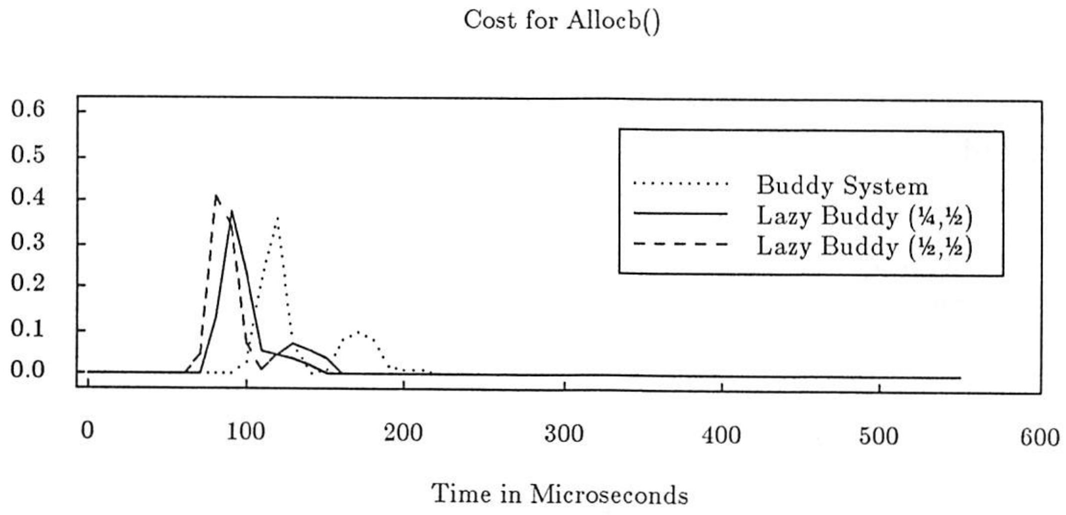


Figure 4. Probability Density Functions for Exponential Interarrival and Service Times

3.1.2 Stochastic Scenarios

Given the average statistics, we can generate three types of workloads with different stochastic structures. The first workload scenario is the D/D/m[10] queue that assumes, for each class of buffers, deterministic arrivals, deterministic service time, and multiple servers. In this context, a *server* is a buffer and *service time* corresponds to the buffer holding time. This type of workload is frequently used for testing and evaluating memory management systems. The second workload scenario is the M/M/m queue that assumes Poisson arrivals and exponential service times. If the goodness of the lazy control policy hinges on deterministic or steady-state usage, the M/M/m workload will allow us to analyze the policy in a non-optimal environment. The third workload scenario has the same M/M/m structure, but is modified to have batched arrivals.[3] This workload is an approximation of the traffic observed in database transaction systems; the buffer requests arrive in bursts corresponding to transactions in the database applications.



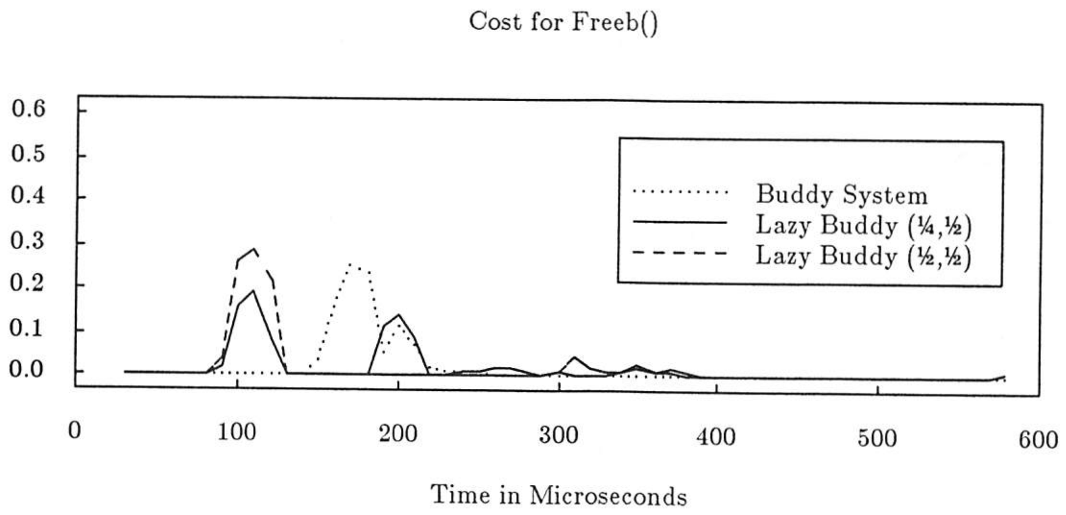


Figure 5. Probability Density Functions for Batched(4) Interarrival and Service Times

3.1.3 Performance Measures

Performance metrics for memory management systems in our work are the mean and standard deviation of the costs to allocate or free a buffer and the allocation failure rate

3. The service times for customers arriving in the same batch are not the same, but are drawn from the same distribution.

(blocking probability). We show these statistics for the three memory managers under the different workloads, and also chart the distributions of these costs for the buddy and lazy buddy systems. All measurements were made on an AT&T microcomputer running UNIX® System V Release 3.2.

3.2 Measurements and Results

We present the results for two sets of policy parameters, (¼,½) and (½,½). The (½,½) is the "laziest" policy, while the regular buddy system corresponds to a (0,½) policy, which coalesces most aggressively.

The aggregate cost distributions comparing the *allocb*() and *freeb*() costs of the buddy system, lazy buddy (¼,½), and lazy buddy (½,½) are shown in Figures 3, 4, and 5; for the batched arrivals, the size of the batch is 4. It is interesting to see the secondary spikes indicating recursive coalescing in the distributions for *freeb*() for both the buddy and the lazy buddy systems.

The means and the standard deviations for the three workloads for the buddy and lazy control policies are summarized in Table 1.

TABLE 1. Means (Standard Deviations) for *allocb()* and *freeb*() Calls (in microseconds)

Memory Policies		Workload Scenarios		
		D/D/m	M/M/m	M/M/m Batch(4)
allocb ()	Buddy System	117.0 (15.8)	110.9 (31.9)	136.8 (35.5)
	Lazy Buddy (¼,½)	90.0 (10.0)	96.5 (18.0)	103.2 (28.3)
	Lazy Buddy (½,½)	82.1 (10.4)	86.9 (15.5)	91.5 (26.5)
freeb ()	Buddy System	160.0 (14.4)	173.0 (25.3)	183.9 (31.0)
	Lazy Buddy (¼,½)	106.7 (12.1)	147.5 (70.4)	184.6 (92.2)
	Lazy Buddy (½,½)	106.3 (12.5)	131.1 (72.8)	150.4 (92.3)

TABLE 2. Means for *allocb*()/*freeb*() Transactions (in microseconds)

Memory Policies	Workload Scenarios		
	D/D/m	M/M/m	M/M/m Batch(4)
Static Allocation	161.2	160.4	162.7
Buddy System	277.0	283.9	320.7
Lazy Buddy (¼,½)	196.7	244.0	287.8
Lazy Buddy (½,½)	188.4	218.0	241.9

TABLE 3. Blocking Probability Statistics

Memory Policies	Workload Scenarios		
	D/D/m	M/M/m	M/M/m Batch(4)
Static Allocation	0.0	0.0290	0.104
Buddy System	0.0	0.000135	0.0218
Lazy Buddy (¼,½)	0.0	0.000257	0.0266
Lazy Buddy (½,½)	0.0	0.000257	0.0328

The costs of *allocb*() are uniformly cheaper with the lazy buddy systems. The costs of *freeb*() are extremely good for the deterministic workload and compare reasonably in the batched case since systems with batched arrivals do not exhibit slow time-varying buffer demands. The differences in *freeb*() costs are more dramatic than those in *allocb*() costs since most of the overhead is absorbed in the release and coalesce phase. Also as expected, we have larger standard deviations for the *freeb*() operations in the M/M/m and M/M/m Batch(4) cases.

If we consider the combined *transaction* cost of an *allocb*() and a *freeb*() pair, shown in Table 2, the lazy buddy system is uniformly better than the buddy system. For the (¼,½) policy, this ranges from 29% to 14% to 10% for D/D/m, M/M/m, and M/M/m Batch(4), respectively. For the (½,½) policy, the corresponding improvements are 32%, 23%, and 25%. The statistics of the original static STREAMS buffer allocation are included in Tables 2 and 3 for comparisons.

Table 3 shows the differences in blocking probabilities among four different policies. As expected, the static policy suffers most when the workload is bursty; the lazy control policies are only slightly worse than the buddy system.

4. DISCUSSION

The initial lazy buddy prototype was intended to provide a dynamic memory manager for STREAMS so that we could take measurements. The largest request size that the prototype would honor was for 4 KBytes; this was because the largest buffer that STREAMS will request is 4 KBytes, and because binary buddy systems tend to waste memory due to internal fragmentation if the buddies become too large. In the full implementation as a general purpose kernel memory allocator, however, we wanted to allow any size request. For requests larger than 4 KBytes we do not use the lazy buddy system. Instead, we use kernel utilities that manage kernel virtual memory to allocate pages to satisfy the request. These allocations are more expensive, but the system seldom requests and releases large pieces of dynamic memory.

The kernel memory allocator presented in [11] is similar to our implementation in that it also uses a powers of two memory allocator for small requests and a kernel virtual memory allocator for large requests. However, the lazy buddy system presented here emphasizes coalescing rules that allow all buffer space to be completely shared. A memory allocator without a coalescing strategy is susceptible to bursty demands that leave a long chain of buffers available only to a particular request size.

We also investigated the use of the "fast fits" algorithm based on a Cartesian tree data structure as described in [12]. Measurements similar to the ones made on the buddy and lazy buddy systems showed that the algorithm was not fast enough for a kernel-level memory manager. Also, allocations with the fast fit algorithm were particularly expensive. The lazy buddy policy, on the other hand, has inexpensive and relatively deterministic costs for allocating buffers. This is a desirable property for time-critical kernel activities such as interrupt service routines that request dynamic memory.

The measurement results presented in this paper were obtained in a carefully controlled environment and with a synthetic memory workload so that we could fairly compare the different systems. Operational costs measured for the lazy buddy system under a live load are similar to the costs seen with the D/D/m workload. Additionally, under a live load the system generally shows 40%-60% memory utilization. This efficiency measure, defined as the ratio of the requested amount of memory over the required amount[13], is considered good for a dynamic memory manager.

5. *CONCLUSIONS*

The watermark-based lazy control policy of the buddy system for UNIX kernel memory allocation is a balance between low operational costs and low blocking probabilities. On the one hand, it approaches the cost structure of the static memory policy for allocating and releasing buffers. On the other hand, it achieves a low blocking probability similar to that of the buddy system. The policy ensures that all memory space will be available to any buffer class and guarantees bounds on the cost to allocate or free a buffer. The lazy control policy also demonstrates a self-metering and self-policing strategy that distributes control over each class of buffer size; distributed control is particularly attractive when the costs of maintaining global information and control are prohibitive.

The measurement results of the prototype system shows that with slow time-varying memory demands, the lazy control achieves its design goal to be low both in operational costs and blocking probabilities.

6. *ACKNOWLEDGEMENTS*

We would like to thank Don Milos and Nancy Mintz for their contributions to this study.

REFERENCES

1. D. E. Knuth, *The Art of Computer Programming, Vol I, Fundamental Algorithms,* Addison-Wesley, Reading, Mass. 1968.
2. J. L. Peterson and T. A. Norman, "Buddy Systems," *Commun. of the ACM,* Vol. 20, No. 6, June 1977, pp. 421-431.
3. T. P. Lee and R. E. Barkley, "Configuring UNIX STREAMS Communication Buffers Based on an Erlang Traffic Model," Proc. of the 1988 Computer Networking Symposium, April 1988, pp. 230-234.
4. T. P. Lee and R. E. Barkley, "A Two-Threshold Lazy Coalescing Control Algorithm for Buddy Systems," paper submitted for publication.
5. *UNIX System V STREAMS Primer,* AT&T Select Code 307-229, 1986.
6. *UNIX System V STREAMS Programmer's Guide,* AT&T Select Code 307-227, 1986.
7. S. Gaede, "A Scaling Technique for Comparing Interactive System Capacities," *Conference Proc. of CMG XIII,* December 1982, pp. 62-67.
8. R. E. Barkley and T. P. Lee, "A Heap-based Callout Implementation To Meet Real-Time Needs," Proc. of the 1988 Summer USENIX Conference, June 1988, pp. 213-222.
9. R. E. Barkley and D. Chen, "CASPER the Friendly Daemon," Proc. of the 1988 Summer USENIX Conference, June 1988, pp. 251-260.
10. L. Kleinrock, *Queuing Systems Volume I: Theory,* John Wiley & Sons, 1975.
11. M. K. McKusick and M. J. Karels, "Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel," Proc. of the 1988 Summer USENIX Conference. June 1988, pp. 295-303.

12. C. J. Stephenson, "Fast Fits: New Methods for Dynamic Storage Allocation," Proc. of the Ninth ACM Symposium on Operating Systems Principles, October 1983, pp. 30-32 (extended abstract)

13. D. Korn and K. P. Vo, "In Search of a Better Malloc," Proc. of the 1985 Summer USENIX Conference, June 1985, pp. 489-506.

TOWER STREAMS-Based TTY: Architecture and Implementation

Mark D. Campbell, m.campbell@ncrcae.Columbia.NCR.COM
Tracy R. Edmonds, t.edmonds@ncrcae.Columbia.NCR.COM

NCR Corporation
West Columbia, South Carolina 29169

Abstract

This paper discusses architectural and implementation issues of STREAMS-based TTY with respect to the members of the TOWER family. Issues examined include general architectural concerns, implementation decisions, performance, and efficiency with respect to unintelligent, semi-intelligent, intelligent, and multi-noded intelligent TTY subsystems. Three classes of STREAMS-based TTY architectures, local, remote, and hybrid STREAMS-based techniques, are examined and critiqued with respect to the four types of TTY subsystems present on TOWER systems today. The TOWER STREAMS-based TTY subsystem (hereafter referred to as TSTTY) is presented in light of these models.

Performance and system efficiency data of the Clist- and STREAMS-based TTY subsystems are given for the semi-intelligent and intelligent TTY subsystems. The myth of increased character-level response time associated with STREAMS-based TTY is disproven with results showing that the TSTTY scheme actually results in the decrease of character-level response time.

1. Introduction

AT&T has been slowly migrating their operating system base towards STREAMS-based TTY and has built several products around the concept, the most notable being their Native Language System (NLS) and the V.3.2 pseudo-terminal mechanism. UNIX^{TM}† System V.4 is rumored to support only STREAMS-based TTY and to provide functionality for both the System V and BSD line disciplines. It currently appears that support of STREAMS-based TTY will be necessary to support many of the features available in future releases of UNIX.

Given that the future of TTY support in the UNIX operating system appears to lie in the direction of STREAMS support, the obvious problem is the migration of all TOWER family members to a STREAMS-based TTY solution. Therein lies the problem -- the TOWER family is made up of three major processor/memory platforms and supports four basic types of TTY subsystem platforms. The four basic types of TTY subsystems supported in the TOWER family are listed below:

- *Unintelligent TTY Subsystems.* This type of TTY subsystem is made up simply of one or more UART devices. An unintelligent TTY subsystem may be characterized as one in which each received or transmitted character results in the host processor receiving an interrupt regardless of the canonicalization mode of the TTY channel. An example of this type of TTY subsystem is the remote diagnostic and console ports of the TOWER 32/650 and TOWER

† UNIX is a registered trademark of AT&T

32/850.

- *Semi-Intelligent TTY Subsystems.* This type of TTY subsystem is made up of one or more UART devices along with a minimal amount of hardware designed to offer some degree of host processor off-loading. An example of this type of TTY subsystem is given by the TOWER 32/200 in which 2 DUART's are associated with custom DMA hardware and a split-level interrupt scheme to attempt to off-load the host processor as much as possible.
- *Intelligent TTY Subsystems.* An intelligent TTY subsystem is made up of an auxiliary I/O processor, support hardware, and UART devices which communicates with the host processor via some well-defined interface. The major differentiation between this type of TTY subsystem and an intelligent TTY subsystem is simply a matter of the degree of off-loading associated with the TTY subsystem. The metric which we use to distinguish these two types of TTY subsystems is whether or not an auxiliary I/O processor is used; if so, then the TTY subsystem is intelligent. An example of this type of TTY subsystem is the HPSIO (**H**igh **P**erformance **S**erial **I**nput/**O**utput) subsystem used on all Multibus-I-based TOWER family members and the TP (**T**erminal **P**rocessor) subsystem used on all Multibus-II-based TOWER family members.
- *Intelligent Multi-Node TTY Subsystems.* An intelligent multi-node TTY subsystem is made up of two or more auxiliary I/O processors with the UART devices controlled directly by the auxiliary I/O processor furthest away from the host processor. Examples of this type of subsystem are the CCHP (**C**luster **C**ontroller **H**ost **P**rocessor) found on Multibus-II-based TOWER systems and the DTCS (**D**istributed **T**erminal **C**ontroller **S**ubsystem) found on Multibus-I-based TOWER systems. Both are made up of a Multibus-II-/Multibus-I-based auxiliary I/O subsystem which connects via a network to one or more auxiliary I/O subsystems which in turn support some number of UART-based RS-232C channels.

On semi-intelligent, intelligent, and intelligent multi-node TTY subsystems NCR has made a major hardware and software investment to solve the problems associated with the efficient support of the UNIX TTY subsystem. There are two major components of TTY subsystem efficiency which have received particular attention: the input and output performance on a per-channel basis and the host processor degradation due to the handling of received and transmitted characters. This past work led to a set of stringent requirements for all TSTTY subsystems. These requirements are listed below:

- It can not significantly increase the percentage of host CPU utilization due to the processing associated with the TTY subsystem.
- It can not reduce the current level of TTY performance on any TOWER family member.
- It must be 100% backward compatible with previous releases of TOWER TTY subsystems. Existing applications must execute as they did on the older Clist-based versions of TOWER TTY subsystems.
- It must provide a homogeneous interface to applications and users regardless of the hardware TTY subsystem on which it is executing.
- It must offer all diagnostic facilities found on current TOWER TTY subsystems. These diagnostic facilities must be common across all hardware TTY subsystems. Examples of these diagnostic facilities include common diagnostic *ioctl(2)*'s, loop-back diagnostic support, and low-level (at the UART itself) data capture.
- It must offer at least the same degree of fault tolerance as current TOWER TTY subsystems.

Architecting and implementing a new TTY subsystem which meets all of these requirements was difficult. The various trade-offs among these requirements are discussed in detail in the remaining chapters of this document.

2. TSTTY Software Architecture

This section describes the software architecture of TSTTY. All TSTTY drivers are based upon this reference model. This model and the corresponding components of the TSTTY driver are examined in the following subsections.

2.1 TSTTY Reference Architecture

The TSTTY subsystem is based upon this reference model and AT&T's *Termio* standard. The TSTTY Reference Architecture reference model is shown in figure 1. The lowest level of this model, the physical layer, is responsible for the transmission of raw data over the physical medium. The physical medium is constituted by the wires which which make up the actual RS-232C connection.

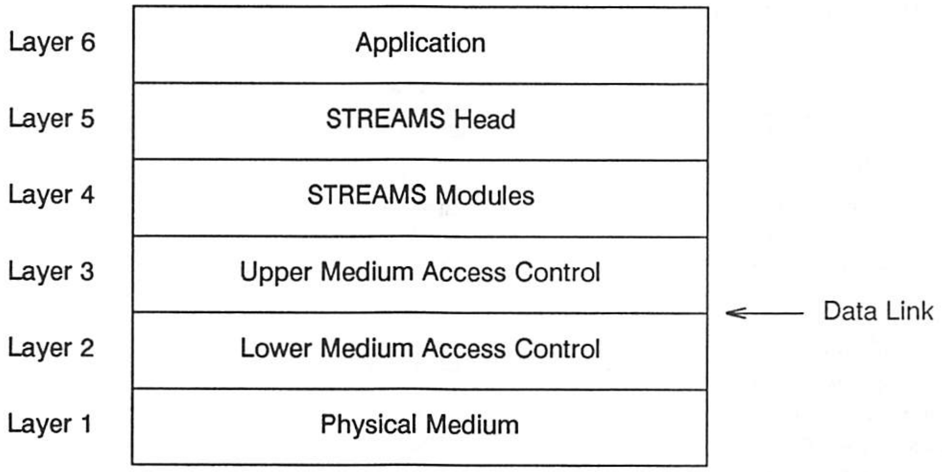


Figure 1. TSTTY Reference Architecture

The next two levels of this model make up the upper and lower Medium Access Control layers. The lower MAC is the device dependent layer. In an unintelligent or semi-intelligent TTY subsystem this layer handles direct manipulation of the UART device; in an intelligent or multi-noded intelligent TTY subsystem the lower MAC consists of code located on the host as well as code located on the auxiliary I/O controller. The upper MAC and all layers above it are completely device independent and in fact exist as common code across the TOWER family members. The upper and lower MAC together are referred to in the *AT&T Programmer's Guide* as the *STREAMS Driver.* The upper MAC consists of the actual STREAMS interface to the device dependent lower MAC.

The MAC layers are minimally responsible for handling what *Termio* defines as *Hardware Control*, as well as the packetizing and statistical multiplexing of receive and transmit data to reduce the number of interrupts to the CPU. The MAC layers may optionally take on the full functionality defined by *Termio* including line discipline and special character handling by implementing the lower MAC accordingly and without altering the upper MAC in any way. Examples of this full *Termio* functionality would be the Remote STREAMS Modules support support discussed later in this paper. The upper and lower MAC layers communicate very efficiently through a well-defined NCR proprietary interface.

The STREAMS Modules Layer is that layer in which all pushed STREAMS modules reside. The primary STREAMS module is the line discipline module. The line discipline STREAMS module is minimally capable of handling what *Termio* defines as *Output Control*, *Line Discipline*, and

Special Control Characters. This includes features such as character echoing, character mapping, canonicalization, and interrupt key handling. The line discipline is a generic pushable STREAMS module and has the capability of turning on and off various aspects of character processing.†

Other STREAMS modules may also exist in the STREAMS Modules Layer. The best example of this is the *sxt* STREAMS multiplexing module which implements the *Shell Layers*-associated functions.

The STREAMS Head Layer is simply the generic STREAMS head section of the kernel with all of the modifications made by NCR to decrease response time latency, increase performance, and maintain backward compatibility with past TTY subsystem releases. It is to this layer that the user application directly interfaces.

In the remaining sections of this chapter the TSTTY reference architecture will be examined from the standpoint of unintelligent, semi-intelligent, intelligent, and multi-noded intelligent TTY subsystems. Different implementations of the reference model upon these types of TTY subsystems will be discussed.

2.2 Unintelligent and Semi-Intelligent TTY Subsystems

The only type of STREAMS approach available on unintelligent or semi-intelligent TTY subsystems is local STREAMS modules support. This technique allows the host processor to push and pop STREAMS modules which lie between the application and the code handling the UART within the host processor's directly addressable memory.

The disadvantage of this approach is that the host processor must handle almost all TTY-related interrupts and must execute all TTY subsystem software. This disadvantage is present on both the Clist-based and the STREAMS-based TTY subsystem; the only difference between the two being attributable to the amount of code which must be executed for either type of TTY subsystem and the maximum number of bytes which may be transferred by the TTY subsystem at any one time on a DMA-based TTY subsystem.

An example of the TSTTY reference architecture on a semi-intelligent TTY subsystem can be given from the TOWER 32/200 TTY subsystem. In this subsystem the lower MAC interfaces directly to the Z8530 DUART's and to the support logic associated with these devices. The STREAMS Modules Layer is made up of the STREAMS line discipline module which is pushed at channel primary initialization (via *init* or *getty*).

The section of this paper entitled *TTY Performance and System Efficiency* gives quantitative results for the TOWER 32/200 semi-intelligent TTY subsystem. The results of this chapter are that the TSTTY subsystem is slightly higher in performance while impacting system efficiency less than the previous TOWER 32/200 Clist-based TTY subsystem.

2.3 Intelligent and Multi-Noded Intelligent TTY Subsystems

There are two basic STREAMS approaches which may be used with intelligent and multi-noded intelligent TTY subsystems: support for local STREAMS modules or support for remote STREAMS modules. Both techniques are examined and critiqued in the following two

† In the case of hybrid STREAMS modules support, the line discipline exists as distinct software on the auxiliary I/O subsystem. This is a degenerate form of a fully remote STREAMS module scheme in which the line discipline would be remotely pushed onto the auxiliary I/O subsystem. This is discussed in detail later in this paper.

subsections. After these basic STREAMS approaches are discussed a hybrid model is presented which lies between the two in terms of performance, efficiency, and complexity. This hybrid model is that upon which all intelligent TSTTY subsystems are based.

2.3.1 Local STREAMS Modules Support

Local STREAMS modules support refers to the ability by a host processor to push and pop STREAMS modules locally, i.e., at the host processor. Using this technique the role of an intelligent auxiliary I/O subsystem is reduced from a subsystem handling various forms of canonicalization to an intelligent DMA subsystem having at best the ability to reduce interrupts via a time-based statistical multiplexing packetizing approach. This approach increases both the average number of interrupts the host processor must handle as well as the amount of code that the host processor must execute on behalf of the TTY subsystem.

In order to minimize the number of interrupts which the host processor must handle with a local STREAMS modules approach it is possible to build a control block interface between the host and auxiliary processors in which real-time tuneable parameters are passed down from the host processor to the auxiliary I/O processor specifying the minimum and maximum amount of time which may elapse from the time a character is received by the auxiliary I/O processor until that character is transmitted to the host processor. Transmission should be handled in such a manner as to mitigate the number of interrupts associated with transferring the characters out of the UART -- in essence, an intelligent DMA scheme. This adds very little complexity to the code executing at the auxiliary I/O processor and provides quite a bit of interrupt handling off-loading under pathological I/O conditions.

The obvious benefit of local STREAMS modules support is its simplicity. The amount and complexity of the code which is executed on the auxiliary processor is kept to a minimum. Given the maintenance and reliability problems associated with functionally partitioning the UNIX TTY subsystem, this approach offers the decided advantage of tremendously lower development and maintenance costs when compared to the remote STREAMS modules support technique.

2.3.2 Remote STREAMS Modules Support

Remote STREAMS modules support refers to the ability by a host processor to push and pop STREAMS modules onto some type of intelligent auxiliary I/O processor. This capability allows host software to configure the auxiliary I/O subsystem in real-time to perform different types of canonicalization. Thus this technique would allow a functional rather than time-based scheme for reducing not only the number of interrupts the host processor must handle but also the TTY subsystem code that the host processor would have to execute.

Remote STREAMS modules support is the great unexplored frontier of all STREAMS-based drivers. Unfortunately, as in almost all unexplored frontiers, the cost of achieving an objective is increased by the cost of forging the trail. The remote STREAMS approach was rejected in the case of the TSTTY subsystem not only because of the high development cost but also because of reliability and maintainability concerns. Even using an existing operating system such as XINU and adding STREAMS recognition was deemed overly complex.

2.3.3 A Compromise Technique -- Hybrid STREAMS Modules Support

Local STREAMS modules support offers simplicity and a corresponding ease of development and maintenance at the cost of a degradation of overall system efficiency. Remote STREAMS modules support offers increased overall system efficiency but has tremendous development and maintenance penalties. The dichotomy between the two mechanisms is obvious.

Fortunately there appears to be a solution which offers many of the system efficiency benefits of remote STREAMS support with only slightly more associated development and maintenance costs than that associated with local STREAMS support. This has been termed *hybrid STREAMS modules support* and is the base architecture for TSTTY. In this approach a subset of the functions performed by the pushable line discipline module are performed by software which executes on one or more auxiliary I/O processors. For example, a set of special characters and a timeout value may be sent to an auxiliary I/O processor with a command which specifies that for a specified channel no interrupts are to be sent to the host processor until one of those special characters is received or the timeout value expires.

This solution is completely generic in that the line discipline can be turned off from the host by sending the appropriate command downstream to the auxiliary I/O subsystem. The host system can then locally push a generic line discipline onto the TTY Stream. At this point the overall system then appears to support simply local STREAMS modules. The degenerate form of hybrid STREAMS support is simply local STREAMS support. It is through this mechanism that common pushable line discipline modules are maintained -- pushable line discipline modules handle unintelligent and semi-intelligent TTY subsystems simply as intelligent TTY subsystems with all auxiliary I/O processing turned off.

The primary difference between hybrid STREAMS Modules support and remote STREAMS support is that there is no mechanism in place to push and pop entire STREAMS pushable modules onto the auxiliary I/O controller dynamically. Instead, the pushable line discipline which executes on the system processor is modified to recognize that certain functions that it must execute in a unintelligent or semi-intelligent can now be executed on the processor(s) of the intelligent or multi-noded intelligent TTY subsystem.

Using this technique canonicalization can occur at the behest of the auxiliary I/O processor without using any host processor cycles; thus many of the advantages of the functionally partitioned TTY subsystem can be obtained. The obvious drawback to this approach is the additional development which must be performed to modify both the pushable line discipline executing upon the host and the support for I/O primitives which must be added in the auxiliary I/O processor. These modifications are minor when compared to the tremendous gains in efficiency which are achieved using this approach.

3. TTY Performance and System Efficiency

It is a truism in computer design that the genericization of some component tends to decrease the performance of that component. From that standpoint the greatest strength of STREAMS-based TTY, the ability to handle widely varying types of TTY input and output, would seem to be its downfall as well. Fortunately all prototypes of STREAMS-based TTY drivers which we have implemented to date exhibit better performance and efficiency than Clist-based drivers on unintelligent and semi-intelligent TTY subsystems. The performance and efficiency penalties which are normally associated with genericization are more than offset by the more efficient algorithms used in TSTTY.

3.1 TTY Performance

There are two types of TTY performance which are commonly measured: transmission performance and receive performance. Of these, transmission performance is the most easily measured and for this reason is the most often measured. In either the Clist- or the STREAMS-based TTY subsystem, however, it is increased receive performance which is the far more difficult to obtain with either a cost-effective hardware or software TTY subsystem solution. Both types of performance are discussed in the sections that follow.

3.2 TTY Transmit Character Performance

In terms of strict TTY performance all TSTTY implementations have been implemented show an increase in TTY multi-character transmission performance (see table 3). One reason for this is that the 64-character Clist (128-character in the case of the TOWER 32/200) has been replaced by the more flexible STREAMS buffer. STREAMS buffers are available from 4-bytes to 4096-bytes in length; thus longer transfers tend to be somewhat more efficient since the size of the transfer can attain a greater size.

Due to the potentially larger buffer sizes of STREAMS-based TTY all of the previously mentioned types of TTY subsystem fair very well with respect to transmission performance. One reason for this is that the time spent allocating and freeing resources is decreased significantly. The Stream head can allocate enough contiguous buffer space to handle almost any application's TTY write request in a single resource allocation call and hence move the data from user space to system space more efficiently. This savings is then passed to modules/drivers downstream which will also spend less time and host processor cycles allocating and freeing buffer resources. Semi-intelligent, intelligent, and multi-noded intelligent TTY subsystems exhibit an additional increase in transmission performance due to the ability of these subsystems to transfer characters from host memory to the UART device without the overhead associated with per-character interrupt processing.

3.3 TTY Receive Character Performance

There are two metrics which we currently use to characterize TTY receive performance. The first metric is the length of time that the front-level ISR which reads from the DUART's FIFO uses. This metric is important because it determines the maximum receive performance of any TTY subsystem in a steady-state condition ignoring character receive transients. For example, if the character receive ISR of the TTY subsystem lasts 1ms, then at 9600 baud only one channel can be supported in a steady-state receive condition. If two channels receive characters at 9600 baud in steady-state, then eventually the FIFO of the UART will be overrun and hardware flow control will be invoked. Table 1 shows a number of receive character ISR times and the associated number TTY channels which can be supported at a number of bauds.

	20us	40us	62.5us	125us	250us	500us	1ms
4800 Baud	96	48	32	16	8	4	2
9600 Baud	48	24	16	8	4	2	1
19200 Baud	24	12	8	4	2	1	0
38400 Baud	12	6	4	2	1	0	0

TABLE 1. Receive Character ISR Times Versus Maximum Concurrent Performance

The original development port of the AT&T UNIX System V.3.1 Clist-based TTY subsystem to the TOWER 32/200 had a receive character ISR time of just under 1ms. Almost all of this time was spent in the line discipline code. This was deemed as unacceptable soon after this port.

In order to increase the receive character performance of the TOWER 32/200 the TTY subsystem was split into two ISR's: a front-end TTY ISR which simply reads all characters in the FIFO of the UART, places those characters in a memory buffer, and asserts a very low-level programmable interrupt. The programmable interrupt ISR reads from the memory buffer and executes the line discipline code. Using this technique the current front-end TTY ISR time on the TOWER 32/200 STREAMS-based TTY subsystem has been reduced to 25us.

Note that this technique is applicable to all of the various classes of TTY subsystems since the key to it is the rapid clearing of the FIFO of the UART in order to prevent hardware flow control from being asserted. This technique is being used where possible on all TSTTY subsystem implementations.

The second metric is the length of time spent in processing the character before it is transferred to the user. This metric is important because it aids in determining the maximum receive throughput associated with different packet sizes of received characters. The table below gives a comparison of the amount of time required to process different size packets of characters for Clist and STREAMS type line discipline schemes.

Characters/Packet	1	2	4	8	16	32
Clist Version	579.2us/char	415.3us/char	315.7us/char	248.3us/char	211.9us/char	193.8us/char
STREAMS Version	468.1us/char	317.2us/char	253.0us/char	216.9us/char	198.4us/char	189.4us/char

TABLE 2. Input Processing Time For Line Disciplines

These numbers are based upon TOWER 32/200 release 1.02 STREAMS-based TTY and upon TOWER 32/200 release 1.01 Clist-based TTY. The numbers show the STREAMS-based line discipline to be on average 14% more efficient than the Clist-based version. More importantly in the case of intelligent TTY subsystems, these results show the dramatic improvement which can be achieved via packetization. Thus even in raw mode the host processor overhead can be significantly reduced by using the auxiliary I/O processor to packetize all of the characters being received.

TTY performance in terms of response time is slightly more complicated. Due to the attention it has received throughout the industry, a subsection devoted solely to it is presented.

3.3.1 The Great Myth -- Drastically Increased Response Time

A common complaint against STREAMS-based TTY is that response time is noticeably increased such that under heavy loads users may perceive a lag time between the time a key is typed and the resultant echoing due to the received character. This is usually explained as being the result of the overhead which STREAMS processing places on both of the basic types of TTY subsystem character processing.

There are two modes associated with TTY subsystem processing: raw and canonicalized character processing. In raw mode a received character is passed from the driver to the application with only limited TTY driver handling. In canonicalized mode a received character may be manipulated by the driver in some fashion and then buffered before it is received by the application. Raw mode character processing is requested by applications such as *vi*, *VA*, and *Multiplan* while canonicalized processing is requested by *sh* and most utilities.

The actions that an unintelligent TTY subsystem takes upon the receipt of a character in raw mode for a representative TOWER STREAMS- and Clist-based TTY driver are shown below in figure 2. Please note that these actions are those associated with the TOWER pushable line discipline module which has been enhanced to decrease the response time associated with receiving characters.

STREAMS-Based TTY Driver	Clist-Based TTY Driver
High-Level Receive ISR Is Asserted.	High-Level Receive ISR Is Asserted.
ISR Reads All Characters From DUART's FIFO.	ISR Reads All Characters From DUART's FIFO.
ISR Puts Characters Into Static Receive Buffer.	ISR Puts Characters Into Static Receive Buffer.
ISR Asserts Low-Level Programmable Interrupt.	ISR Asserts Low-Level Programmable Interrupt.
Low-Level Receive ISR Is Asserted.	Low-Level Receive ISR Is Asserted.
ISR Reads All Characters from Static Receive Buffer.	ISR Reads All Characters from Static Receive Buffer.
ISR Gets a STREAMS Buffer.	ISR Gets a Clist Structure.
ISR Puts Characters Into a STREAMS Buffer.	ISR Puts Characters Into a Clist Structure.
ISR Sends Characters Upstream.	ISR Asserts Line Discipline Via Line Switch Table.
Line Discipline STREAMS Module Is Asserted.	Line Discipline Is Asserted.
The Character Is Sent Upstream.	
STREAMS Head Is Asserted.	
Character Is Copied From Stream Buffer to User Space.	Character Is Copied From the Clist to User Space.
Waiting User Process Is Woken.	Waiting User Process Is Woken.

Figure 2. STREAMS and Clist-Based Raw Mode Processing

The actions that an unintelligent TTY subsystem takes upon the receipt of a character in canonicalized mode for a STREAMS and Clist-based TTY driver are shown below in figure 3. As in figure 2, these actions are those associated with the enhanced TOWER pushable line discipline.

STREAMS-Based TTY Driver	Clist-Based TTY Driver
High-Level Receive ISR Is Asserted.	High-Level Receive ISR Is Asserted.
ISR Reads All Characters From DUART's FIFO.	ISR Reads All Characters From DUART's FIFO.
ISR Puts Characters Into Static Receive Buffer.	ISR Puts Characters Into Static Receive Buffer.
ISR Asserts Low-Level Programmable Interrupt.	ISR Asserts Low-Level Programmable Interrupt.
Low-Level Receive ISR Is Asserted.	Low-Level Receive ISR Is Asserted.
ISR Reads All Characters from Static Receive Buffer.	ISR Reads All Characters from Static Receive Buffer.
ISR Gets a STREAMS Buffer.	ISR Gets a Clist Structure.
ISR Puts Characters Into a STREAMS Buffer.	ISR Puts Characters Into a Clist Structure.
ISR Sends Characters Upstream.	ISR Asserts Line Discipline Via Line Switch Table.
Line Discipline STREAMS Module Is Asserted.	Line Discipline Is Asserted.
Canonicalization Occurs.	Canonicalization Occurs.
If Special Character, Send STREAMS Buffer Upstream;	If Special Character, Continue;
Otherwise Resume System Duties (*rte*).	Otherwise Resume System Duties (*rte*).
STREAMS Head Is Asserted	
Character(s) Are Copied From Stream Buffer to User Space.	Character(s) Are Copied From the Clist to User Space.
Waiting User Process Is Woken.	Waiting User Process Is Woken.

Figure 3. STREAMS and Clist-Based Canonicalized Mode Processing

After reading these two figures it should be obvious that no system level delays (e.g., *sleep()*'s, etc.) are present which would lead to one system being inherently less efficient than the other in terms of response time. In table 2 we saw that TSTTY pushable line discipline times are actually slightly less than Clist-based line discipline times. Thus the response time for TSTTY is actually marginally better than that associated previous TOWER Clist-based TTY subsystems.

3.4 System Efficiency

It was said earlier that TTY performance and system efficiency have long been viewed as being directly at odds with one another. It was then shown that new hardware and software techniques could mitigate the impact of one on another. The key term here is *mitigate* -- as we shall see in this section, there is still a strong inverse correlation between the two.

The impact of STREAMS on the TTY subsystem depends upon whether the subsystem under examination is one which has limited hardware support and thus is unintelligent/semi-intelligent or whether it is one which is truly intelligent and can off-load the host processor from much of the processing related to the TTY subsystem. The impact of STREAMS on these classes of TTY subsystems is discussed in the subsections below.

3.4.1 Unintelligent and Semi-Intelligent TTY Subsystems

STREAMS-based TTY subsystems which are ported to unintelligent and semi-intelligent subsystems show negligible overhead, and in some cases a small improvement in system efficiency, when compared to Clist-based TTY subsystems on the same hardware base. Theoretically, there must be more overhead associated with a STREAMS-based TTY subsystem simply because of the genericization of the STREAMS interface which results in an increased amount of code which must be traversed due to STREAMS. As we saw in table 2, however, this is mitigated in the case of character receives by the greater algorithmic efficiency of the TSTTY pushable line discipline. This is corroborated by table 3, which gives the results of executing the SCB *trun* character transmission benchmark on both Clist- and STREAMS-based TTY subsystem.

TTY Subsystem Type	**Clist**	**STREAMS**	**Clist**	**STREAMS**	**Clist**	**STREAMS**	**Clist**	**STREAMS**
Number of Channels	1	1	2	2	3	3	4	4
9600 Baud								
Characters/Second/Line	928.6	930.6	919.3	930.6	912.0	929.1	914.7	926.7
Characters/DMA Transfer	126.5	33.0	121.4	33.0	119.7	33.0	119.0	33.0
Percent CPU Utilization	6.72	4.1	12.0	5.8	17.77	13.5	23.6	18.4
Normalized Per Character Percent CPU Utilization	**7.2**	**4.4**	**6.5**	**3.1**	**6.5**	**4.8**	**6.5**	**5.0**
19200 Baud								
Characters/Second/Line	1821.7	1859.6	1798.7	1857.3	1783.0	1853.0	1774.7	1848.0
Characters/DMA Transfer	115.4	33.0	112.6	33.0	112.0	33.0	111.3	33.0
Percent CPU Utilization	13.3	8.6	24.6	20.44	36.7	32.1	50.0	42.1
Normalized Per Character Percent CPU Utilization	**7.3**	**4.6**	**6.8**	**5.5**	**6.8**	**5.8**	**7.0**	**5.7**
38400 Baud								
Characters/Second/Line	3510.0	3713.4	3396.2	3704.7	3347.8	3677.3	3224.2	3674.6
Characters/DMA Transfer	115.4	33.0	112.6	33.0	108.8	33.0	101.0	33.0
Percent CPU Utilization	25.2	21.1	48.6	41.7	72.0	61.9	93.2	82.3
Normalized Per Character Percent CPU Utilization	**7.2**	**5.7**	**7.2**	**5.6**	**7.2**	**5.6**	**7.2**	**5.6**

TABLE 3. TOWER 32/200 Clist and STREAMS-Based TTY Performance/Efficiency

These results are given for the semi-intelligent TTY subsystem of the TOWER 32/200. The major characteristic of the TOWER 32/200 with respect to this table is that the TOWER 32/200 TTY hardware supports the DMA output of characters in packets of up to 64K. The implementation of the TOWER 32/200 Clist-based TTY subsystem is such that at most 128 characters may be

DMA'ed at a time, while the implementation of the TOWER 32/200 STREAMS-based TTY subsystem allows DMA of up to 4096 characters at a time. As this table shows, however, the STREAMS-based version of the TTY subsystem is more efficient than the Clist-based version even with approximately 75% smaller DMA packet sizes. There are two reasons for these somewhat incongruous results: an increase in algorithmic efficiency present in the STREAMS TTY subsystem and the more efficient block copies that are performed by the Stream head from user space compared to its Clist-based predecessor.

3.4.2 Intelligent and Multi-Noded Intelligent TTY Subsystems Performance/Efficiency

There are several advantages associated with TSTTY on intelligent or multi-noded intelligent TTY subsystems when compared to unintelligent or semi-intelligent TTY subsystems:

- An intelligent or multi-noded intelligent TTY subsystem can support some form of time-based packetizing and statistical multiplexing.
- An intelligent or multi-noded intelligent TTY subsystem can support scatter transmission of multiple lists of characters in a single transmit command, i.e., these types of subsystems can support transmission from multiple sources to multiple destinations without host processor interaction. This may be contrasted to the ability of the semi-intelligent TTY subsystem to support transmission of packets characters from a single source to a single destination.
- An intelligent or multi-noded intelligent TTY subsystem can support some form of functionally-based off-loading scheme.

All of these features aid in reducing the number of interrupts which must be handled by the host processor and thereby increase the efficiency of the TTY subsystem. Each is discussed at length in this section.

Time-based packetizing and statistical multiplexing of received characters is basically a simple scheme which is used to decrease the number of interrupts which the host processor must handle for a given set of inputs. This scheme is limited by the number of UART devices connected to each auxiliary processor and the time which may elapse between a character be received and some response to that character being transmitted.

Receive character performance is the major weak point of a strictly local STREAMS module TTY subsystem approach; however, table 4 does show that we can expect substantial improvements if the auxiliary I/O subsystem can support efficient packetization. The information displayed in this table was taken from the TSTTY subsystem implemented on the TOWER 32/650 HPSIO.

Characters/Packet	**1**	**2**	**4**	**8**	**16**	**32**
Time (us)	468.1	634.4	1012.0	1735.2	3174.4	6060.8
Time/Character (us)	468.1	317.2	253.0	216.9	198.4	189.4
Time Delta (us)	0.0	150.9	215.1	251.2	269.7	278.7
Improvement (%)	0.0	32.2	46.0	53.7	57.6	59.5

TABLE 4. Receive System Efficiency Increase Due to Packetizing

Scatter transmission is useful and necessary for increased system efficiency since most applications do not take advantage of the benefits of the larger STREAMS buffers by issuing the largest possible write requests. Preliminary tests indicate that scatter transmission of 2 16-byte Stream buffers results in a 33% decrease in host processor utilization and a 10% increase in performance when compared to using two separate commands to transmit the same 2 STREAMS buffers.

Remote STREAMS support, and its degenerate form of hybrid STREAMS support with its functionally-based off-loading scheme, is obviously the most effective mechanism to increase system efficiency in an intelligent or multi-noded intelligent TTY subsystem. The increase in system efficiency is due to two factors: the off-loading of per-character or per-packet receive interrupt handling and the off-loading of the canonicalization of both received and transmitted characters by the auxiliary I/O subsystem. Qualitative results have been encouraging; unfortunately, quantitative results aren't possible without assuming some exact mix of TTY characteristics with respect to canonicalization mode, received versus transmitted characters, etc. We are currently working on a metric which will allow quantitative assessment of different degrees of functional off-loading.

4. Conclusion

TSTTY provides a seamless next generation TTY subsystem for the TOWER family which increases per-channel TTY performance on all of the TOWER TTY subsystems while sacrificing little if any overall system efficiency. The major problem to date with all current implementations of TSTTY is that they do not offer as much canonicalization off-loading of the host processor in intelligent and multi-noded intelligent TTY subsystems. This is being addressed in subsequent releases by providing more canonicalization primitives at the auxiliary I/O subsystem; however, it should be noted that early results of case studies and load analyses are showing a trend towards the use of less driver-level canonicalization and more application-level canonicalization. In any case the support of time-based statistical multiplexing and packetizing tends to offset the lack of more extensive canonicalization facilities.

The initial general release of TSTTY provides only local STREAMS support; however, all subsequent releases on intelligent or multi-noded intelligent TTY subsystems will offer hybrid STREAMS support. An effort is currently underway to provide full remote STREAMS support; however, this appears to be at least a year away from general release. Both hybrid and remote STREAMS support on intelligent systems have been implemented and are currently executing in a laboratory environment.

5. Acknowledgements

There are quite a few people that deserve credit for many of the results given in this paper. The TOWER 32/200 semi-intelligent Clist-based software which was used for comparison in this document was developed primarily by Mary Lee Young. The functionally partitioned HPSIO intelligent Clist-based software was developed by Todd Davis and enhanced by Steve Snipes and Tim Ozor. Most of the information concerning multi-noded intelligent TTY subsystems came from Gary Mack; his work on the functionally partitioned CCHP intelligent Clist-based software formed the basis of many of the issues discussed in this paper.

Tim Ozor served as the bridge between the Clist- and STREAMS-based HPSIO architectures. Tim has productized the STREAMS-based HPSIO code and is exploring new components of hybrid STREAMS-based intelligent TTY subsystems. We would like to thank all of these developers for their time and efforts.

6. References

1. Campbell, Mark D., *A TTY Architecture for a Next Generation TOWER*, NCR Internal Memo, June 5, 1988.
2. McGrath, G. J., Olander, D. J., *The UNIX Transport Interface Specification - Issue 1*, AT&T Information Systems Internal Memo, May 9, 1985.
3. McGrath, G. J., Hytry, T. L., Singh, J. H., *Logical Link Interface: A Proposed Service Interface - Issue 1*, AT&T Information Systems Internal Memo, October 15, 1985.
4. Olander, David J., Israel, Robert K., McGrath, Gilbert J., *A Framework for Networking in System V*, AT&T Information Systems.
5. *AT&T UNIX System V.3.1 Manual Set*, AT&T Information Systems.
6. *AT&T UNIX System System V Japanese Application Environment Product Overview*, AT&T Information Systems.

Out-Of-Band Communication in STREAMS

Stephen Rago

AT&T Bell Laboratories
Summit, New Jersey 07901

ABSTRACT

The STREAMS mechanism in UNIX® System V provides a flexible framework for the development of networking software. However, the existing mechanism makes little provision for the out-of-band communication facilities found in many contemporary protocols. This paper describes enhancements to STREAMS that provide the underlying support for out-of-band communication. It is assumed that the reader is familiar with the STREAMS mechanism.

Introduction

The STREAMS Mechanism [1], first commercially available in UNIX® System V Release 3.0, provides a flexible framework for the development of networking protocols and services [2]. It promotes a layered approach to the development of networking software through the use of dynamically pushable kernel modules that alter messages as they flow in a stream, analogous to the way user-level commands alter data in a pipeline. However, many contemporary networking protocols allow the transmission of out-of-band data. These are data that are transmitted at a higher priority than normal data, out of the band of normal data flow. STREAMS provides little support for out-of-band messages. This paper will discuss enhancements made to the STREAMS subsystem to provide generalized support for out-of-band messages in UNIX System V Release 4.0 (SVR4).

Most protocol specifications do not define the facilities needed to implement out-of-band communication. Nor do they define the interface between the user and the out-of-band communication mechanism. The protocol specifications are purposely vague so that the protocols may be implemented in many diverse environments. Often the specifications don't even mention how the protocol's out-of-band communication facilities are intended to be used.

For example, the Transmission Control Protocol (TCP) [4] specifies support of out-of-band data by allowing packets to be marked *urgent*. As soon as urgent data are received, the receiver goes into *urgent mode*. A point in the data stream is designated as the end of urgent data. When the user has consumed all of the data up to the mark, the user goes back into *normal mode*. The urgent mark may be updated by the reception of further urgent packets while in urgent mode. The receiver is notified of the arrival of urgent data, but how this is accomplished is implementation-dependent. 4.2BSD implementations send SIGURG to the receiving process and allow the data to be delivered either in-line or out-of-band.

An example of how this is used is the "Synch" mechanism used by the TELNET Protocol[5]. The TELNET "Synch" mechanism provides a way to regain control of a process. One end of a TCP connection generates a TCP urgent notification with the TELNET DATA MARK command as the last data octet. Upon receipt of the urgent data notification, the receiving end filters through the data stream looking for "interesting" signals, discarding all other data up to the DATA MARK.

The ISO Transport Protocol Specification [3] requires support for out-of-band data, called *expedited data*. These are intended to be high priority data, taking precedence over normal data. Expedited data are sequenced. The sequence numbers come from a separate name space than that of normal data. This implies that expedited data are not affected by the flow control constraints of normal data transfer. However, expedited data need their own flow control.

Existing STREAMS messages are not sufficient to provide generalized support for out-of-band communication. Messages are placed on a queue in first-in-first-out (FIFO) order, with priority messages placed ahead of normal ones. Service interfaces are implemented primarily with M_PROTO messages, which belong to the class of normal messages. If a module wishes to give one M_PROTO message priority over another, it will have to order its queue manually, through the **insq**() and **rmvq**() routines [6]. This is inadequate, however, because once the messages are passed on to the next entity in the stream, any concept of ordering is lost and the messages revert to being handled in a FIFO manner.

There are two bands of communication in STREAMS: normal and priority. Out-of-band communication can be obtained by using M_PCPROTO messages, which belong to the class of priority messages. By convention, these messages are not affected by normal flow control constraints. When enqueued, they are placed at the head of the queue, after any other priority messages already on the queue. Placing a priority message on a queue always enables the service procedure for that queue. The drawback to using M_PCPROTO messages lies with the fact that their design intent was to provide a mechanism for high priority interface acknowledgements. As such, only one M_PCPROTO message is allowed on the stream head read queue at a time. If an M_PCPROTO message is on the stream head read queue, subsequent M_PCPROTO messages received by the stream head will be silently discarded until the original one is consumed by the user. This is an unsuitable characteristic for out-of-band messages.

If we were to change this characteristic, the system could easily be run out of memory by the repeated generation of M_PCPROTO messages from downstream when a user refuses to read from the stream. Additionally, the reception of out-of-band messages during state transitions could potentially interfere with the state machine for the service interface.

Adding new message types (M_EXDATA, M_EXPROTO, etc.) solves the problem for protocols with two bands of message flow (normal and expedited). However, if a protocol came along with three bands of flow, this solution could not be easily extended, short of adding even more message types. The addition of new message types presents an even bigger problem: existing modules and drivers would not recognize them. The rules for handling unknown message types specify that modules must pass them on and that drivers must free them. So a module that, for example, processes all data messages, would require changes to recognize the new data message types.

The C Word

Most of the design decisions center around one major factor: compatibility with existing source and object files. User-level applications are "sacred cows," so they can not be broken. Both object and source compatibility are maintained. Additionally, compatibility of STREAMS modules and drivers is maintained. In theory, a STREAMS module written and compiled for SVR3.0 will continue to work in SVR4.0, as long as it is "well-behaved." Some examples of misbehavior would be:

- depending on the size of the data block structure,
- using kernel data structures that are not intended to be publicly accessible (like **cdevsw**),
- or relying on the internals of the STREAMS message allocation mechanism (in SVR4, the message allocation mechanism changed entirely).

Often the design is more complex than it needs to be. This is for the sake of compatibility. One such case involves the **queue** structure. Macros (WR(), RD(), and OTHERQ()) exist that depend both on the size of the queue and the fact that the read and write queues are allocated contiguously. So that modules and drivers continue to work, we created an *extended queue* structure, **struct equeue**. While conceptually distasteful, this allowed us to maintain the correct offsets. The extended queue structure is associated with the queue structure by using a private pointer that was only used internally by the STREAMS mechanism. This is discussed further in later sections.

Kernel object compatibility can be broken on systems where expanded types are supported. On these systems, certain kernel **typedefs** are expanded in size (like **dev_t** growing from a **short** to a **long**). Here, we are free to change the size of the queue and the message block. Depending on whether the system supports expanded types, the location of new fields in the data structures may vary. Fields are defined to hide the differences from the modules and drivers. For example, a flag field was added to the message block. On systems where expanded types are supported, the field exists in the message block. On systems where expanded types are not supported, the field exists in the data block because modules and drivers should not need to know the size of the data block. In either case, the conceptual appearance to the modules and drivers is that the flag is part of the message block.

STREAMS Messages

Previous releases have supported messages queued in the priority order shown in Figure 1. This presents a conceptual view of two different bands of message flow within a stream.

tail	normal messages	priority messages	head

Figure 1. Old Message Ordering on a Queue

The general solution to providing an arbitrary (almost) number of bands of message flow within a stream involves associating a priority number, called the *priority band*, with each message. When a message is enqueued, it is placed on the queue as indicated by its priority band, but after any other messages of the same priority already on the queue. To avoid confusion, the terminology is modified as follows: STREAMS messages previously referred to as *priority messages* are now called *high priority messages*. *Normal messages*, also known as *ordinary messages*, belong to priority band 0. High priority messages ignore their priority band; they are the highest priority by virtue of their message type. These have

```
mp->b_datap->db_type >= QPCTL.
```

High priority messages are still placed at the head of the queue, but after any high priority messages already on the queue. Figure 2 depicts the new message queueing. For convenience, the number of bands is limited to 256, i.e. the priority band is stored as an **unsigned char**.

tail	normal messages	priority band 1 messages	priority band 2 messages	...	priority band *n* messages	high priority messages	head

Figure 2. New Message Ordering on a Queue

This design has the advantage that a message may retain its priority ordering as long as it traverses the stream. It is extensible in that it requires no more changes to the STREAMS subsystem if new protocols should ever require more bands of message flow. The flow control parameters of one band are independent of the other bands. Figure 3 depicts the message ordering of expedited data on a queue.

tail	normal (band 0) messages	expedited (band 1) messages	high priority messages	head

Figure 3. Message Ordering for Expedited Data

As mentioned previously, a flag field is also added to the message. One flag, MSGMARK, enables modules and drivers to "mark" a particular message. This was added to support the notion of identifying a mark in the data stream. The sockets module sets the MSGMARK flag to enable users to determine when they have reached the mark (see the SIOCATMARK **ioctl** command in **TCP(4P)** and **sockio(4)**). In addition to recognizing "marked" messages, the stream head will fragment reads around the marked message. In other words, a read will only return data up to the last byte in a marked message, even if there are more messages on the stream head read queue. Successive reads will retrieve any other data on the queue.

For compatibility, the new fields are in the data block. However, on systems with expanded types, the new fields are in the message block. On systems without expanded types, `b_band` is defined to be `b_datap->db_band` and `b_flag` is defined to be `b_datap->db_flag`. Modules and drivers must take care that on systems without expanded types, that the reference count in the data block is not greater than 1 before changing these fields.

Two STREAMS messages require minor modifications to support a multiple band paradigm. These are the M_FLUSH message and the M_SETOPTS message. The M_FLUSH message previously contained a byte that consisted of a bitmask specifying which queues to flush. Another flag, FLUSHBAND, is added to indicate to the module or driver to only flush the messages in the given band. The band to flush is found in the second byte of the message. A new routine, **flushband()**, is added to allow a module or driver to flush the messages in one particular band.

The M_SETOPTS message contains a **stroptions** structure and is used to set various stream head options. A band field, `so_band`, is added and if the SO_BAND flag is set in the flags field, the band field contains the priority band whose flow control parameters are to be updated. This is used in conjunction with the SO_HIWAT and SO_LOWAT flags.

STREAMS Queues

Again, for compatibility on systems without expanded types, the **queue** structure changes are minimized. The pointer, `q_link`, used only internally for STREAMS scheduling, was usurped and replaced by a pointer to an extended queue structure, **struct equeue**. The extended queue contains a new link pointer, a pointer to a list of **qband** structures describing the bands of the queue, and a count of the number of structures in the list. On systems without expanded types, the extended queue structure does not exist. The new fields are added directly to the queue structure.

The **qband** structures describe each band of messages on the queue. So every queue does not have to carry around the weight of 255 **qband** structures (one isn't needed for band 0), they are allocated dynamically when a message is enqueued if there is not already a structure describing the priority band of the message. These structures are freed when the stream is dismantled.

The **qband** structure contains a linked list pointer, a count of the number of bytes on the queue in messages of the corresponding band, pointers to the beginning and end of the messages on the queue in the corresponding band, high and low water marks for the band, and state flags. Figure 4 shows what a queue would look like containing messages from two extra bands on a system without expanded types.

The `qb_first` and `qb_last` pointers are optimizations used to quickly locate the position on a queue where a message will be placed for the given priority band. However, when a message is being put on a queue, if there are not already messages in the same priority band on the queue, then the queue must still be searched to find the appropriate position. If it turns out that this is usually the case, then the two pointers can be removed in a future release without any loss of functionality.

To insulate modules and drivers from changes in the queue architecture, two routines, **strqget()** and **strqset()**, are added to get and set information about a given queue and priority band. With these routines, it becomes unnecessary for modules and drivers to know whether they are dealing with a **queue** structure from a system supporting expanded types or not.

Since the **qband** structures are allocated dynamically, it is possible for the allocation to fail. Hence, **putq()**, **insq()**, and **putbq()** can fail. Previously, no significance was given to the return values from these routines. Now they return 1 on success and 0 on failure. The routines **strqget()** and **strqset()** also may fail because of **qband** allocation failure.

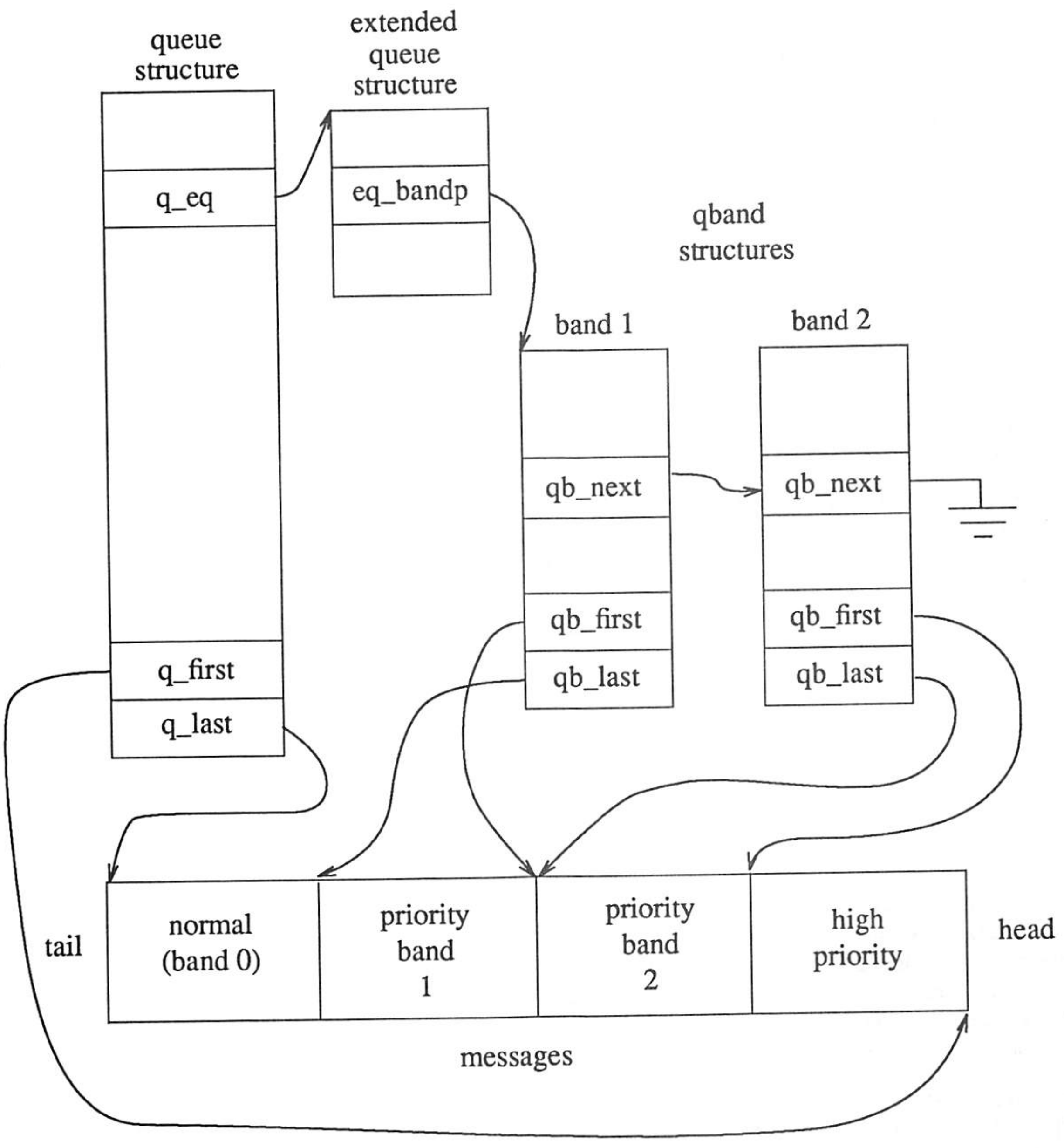


Figure 4. Queue Architecture for Systems Without Expanded Types

On the Inside Looking Out

The world of the module and driver changed very little. Modules and drivers still see one queue with one list of messages. The priority bands merely impose an ordering on the messages on the queue. A module or driver does not need to be written to know specifically about the specified bands. If they wish to do so, modules and drivers are free to set the priority band of any message (except high priority ones) to any valid band value to create the desired ordering.

If they wish to maintain the integrity of the bands, modules and drivers can use a new routine, **bcanput()**, to test for flow control of a particular band. Service procedures are usually written to the following algorithm:

```
while ((bp = getq(q)) != NULL) {
          if (bp->b_datap->db_type >= QPCTL) {
                    /* Process message */
                    putnext(q, bp);
          } else if (canput(q)) {
                    /* Process message */
                    putnext(q, bp);
          } else {
                    putbq(q, bp);
                    return;
          }
}
```

If a module or driver wishes to recognize priority bands, the new algorithm becomes:

```
while ((bp = getq(q)) != NULL) {
          if (bp->b_datap->db_type >= QPCTL) {
                    /* Process message */
                    putnext(q, bp);
          } else if (bcanput(q, bp->b_band)) {
                    /* Process message */
                    putnext(q, bp);
          } else {
                    putbq(q, bp);
                    return;
          }
}
```

It is not required that every module conform to the new algorithm. It only matters when the priority band of messages is significant between two entities (driver, module, or stream head) on the stream, and one or more modules exist between them. Even if an "old" module is interposed between the two neighboring entities that care about message bands, all is not lost. The messages will go from one entity, where they are prioritized on the queue, to the module, where they will be treated in a FIFO fashion on the queue, to the next entity, where they will go back to being prioritized on the queue.

By using **canput()**, the interposing module is causing the message flow to be determined by the flow constraints of normal messages. This loses some of the benefit of out-of-band messages, but it is not catastrophic. It is analogous to sending expedited data over a network connection between two transport endpoints. Both ends of a virtual circuit at the transport level care about the order of expedited data with respect to normal data. At some layer, the expedited data becomes normal data to be transmitted over the communication medium. This will remain normal data until it reaches the first layer that recognizes expedited data. Consider a connection between transport endpoints, without a network layer (see Figure 5). The link layer is connectionless. A transport provider may send a T_EXDATA_REQ to the link layer in the form of an L_UNITDATA_REQ. The expedited data will be treated as normal data and transmitted over the communication medium. The peer link layer will generate an L_UNITDATA_IND. After removing the header, the peer transport layer will find that the data is expedited and create a T_EXDATA_IND to send to the user. It is at this point that the priority nature of the message is actually discovered.

The service procedure algorithm has the effect that if a band is flow-controlled, the higher priority bands are not affected. However, the same is not true for lower priority bands. When a band is flow-controlled, the lower priority bands are also stopped from sending messages. This is necessary to stop lower priority

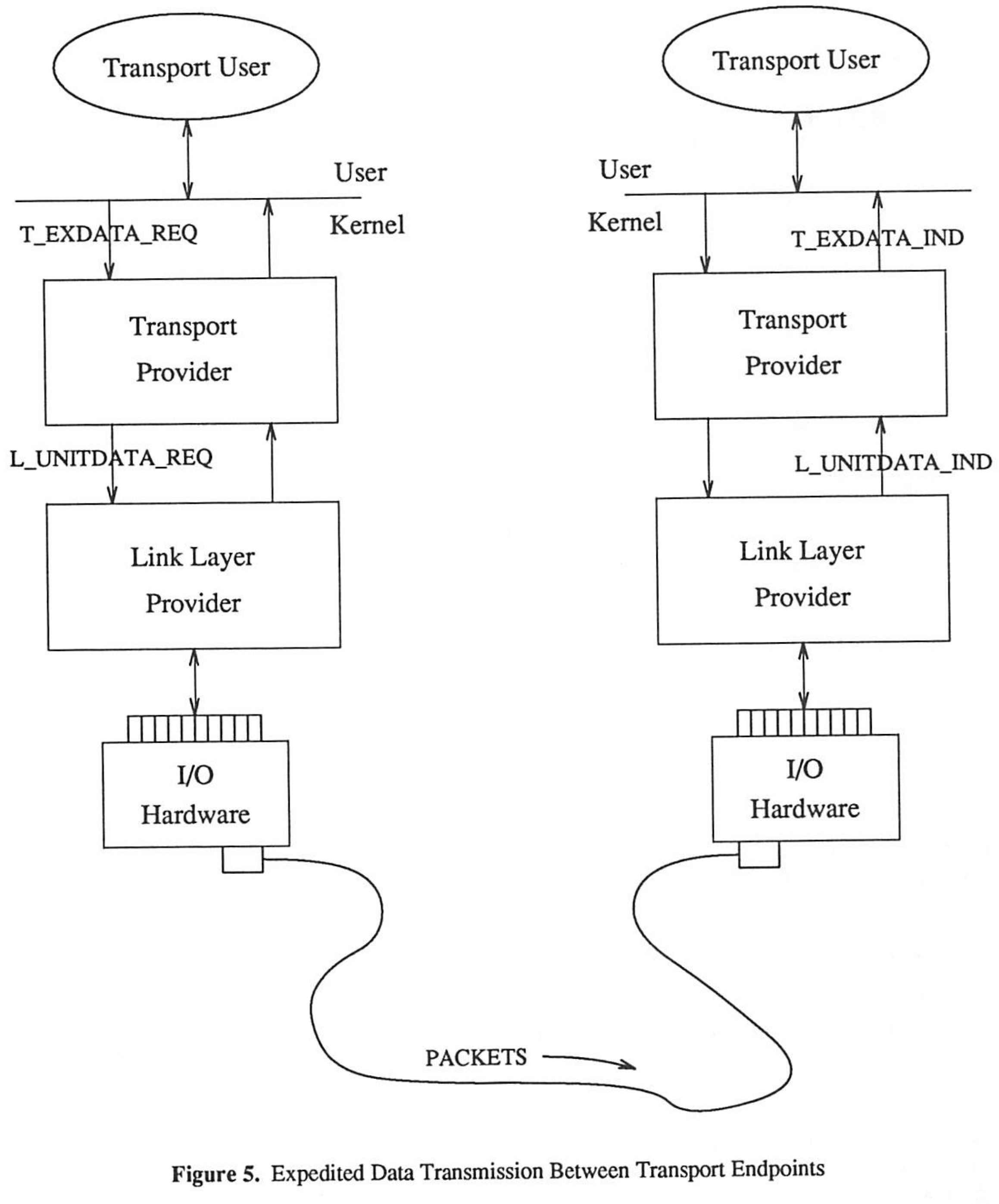


Figure 5. Expedited Data Transmission Between Transport Endpoints

messages from being passed along ahead of the flow-controlled higher priority ones.

By relying on the service interface information rather than the band, modules and drivers can be written to interact properly regardless of the priority band of the messages. For example, in the Transport Provider Interface, a T_EXDATA_REQ would be in band 1, but this is not absolutely necessary since the information identifying the primitive is located in the M_PROTO block.

One other area that is different because of multiple bands is that of queue flushing. As mentioned previously, the M_FLUSH message has been modified slightly. The typical algorithm used to process M_FLUSH messages is:

```
queue_t *rdq;          /* the read queue */
queue_t *wrq;          /* the write queue */

case M_FLUSH:
          if (*bp->b_rptr & FLUSHW)
               flushq(wrq, FLUSHDATA);
          if (*bp->b_rptr & FLUSHR)
               flushq(rdq, FLUSHDATA);
          /*
           * modules pass the message on;
           * drivers shut off FLUSHW and loop the message
           * up the read side if FLUSHR is set; otherwise,
           * drivers free the message.
           */
          break;
```

If a module or driver wishes to recognize priority bands, the new algorithm for flushing becomes:

```
queue_t *rdq;          /* the read queue */
queue_t *wrq;          /* the write queue */

case M_FLUSH:
          if (*bp->b_rptr & FLUSHBAND) {
              if (*bp->b_rptr & FLUSHW)
                     flushband(wrq, FLUSHDATA, *(bp->b_rptr + 1));
              if (*bp->b_rptr & FLUSHR)
                     flushband(rdq, FLUSHDATA, *(bp->b_rptr + 1));
          } else {
              if (*bp->b_rptr & FLUSHW)
                     flushq(wrq, FLUSHDATA);
              if (*bp->b_rptr & FLUSHR)
                     flushq(rdq, FLUSHDATA);
          }
          /*
           * modules pass the message on;
           * drivers shut off FLUSHW and loop the message
           * up the read side if FLUSHR is set; otherwise,
           * drivers free the message.
           */
          break;
```

Again, modules and drivers are not required to treat messages as flowing in separate bands. If they wish, they may still view the queue as before. However, code written the old way will affect the entire queue whenever an M_FLUSH message is received.

What the User Sees

The world of the user has not changed very much. **write(2)** still generates normal M_DATA messages. **read(2)** still obtains whatever is at the front of the stream head read queue. **putmsg(2)** and **getmsg(2)** still work the same. POLLIN events are satisfied by the reception of any data at the stream head, regardless of the band. New events are added for those users that want to distinguish different priority bands. These, along with some **ioctl(2)s**, allow a user to obtain finer control over each priority band. Alternatively, extended signals can be used to get the information regarding the priority band in which the event occurred.

Two new system calls, **putpmsg(2)** and **getpmsg(2)**, are added to allow users to send messages down a particular band and to identify which band the received message came from, respectively. They are similar to **putmsg** and **getmsg**, with minor modifications to accommodate priority bands.

Experiences

As it turns out, when using a sockets module on top of TCP, the sockets module has to subvert much of the mechanism for out-of-band data support. The sockets interface supports two methods of urgent data delivery. The first is in-line with normal data. In this mode, the sockets module has to convert T_EXDATA_IND primitives (implemented as M_PROTO messages in band 1) to T_DATA_IND messages (implemented as M_PROTO messages in band 0). The MSGMARK flag is set in the message containing the urgent mark. The second method of urgent data delivery is to give the data to the user only when explicitly asked. Here, the sockets module holds onto the urgent data until notified by the sockets library routine to send it upstream.

Conclusions

The changes described in this paper are extensible in that protocols are free to define multiple bands of data flow if they so desire, without requiring further changes to the STREAMS mechanism. Each band of message flow has its own independent flow control. The design preserves the module's view of one queue with one list of messages. All messages are linked on the queue based on their priority band of flow. Because of this, modules that do not wish to recognize separate bands of flow will continue to work with those that do.

Acknowledgements

Thanks to the following people who have discussed the issues regarding out-of-band message support in STREAMS: Steve Buroff, Jessica Chan, Larry Feigen, Tom Fritz, Bob Gilligan, Torez Hiley, Gil McGrath, Tamar Newberger, Dave Olander, Marilyn Partel, Dennis Ritchie, Vicki Rosenthal, Art Sabsevitz, Bill Shannon, Glenn Skinner, Walt Tuvell, Ian Vessey, Larry Wehr, and Norman Wilson.

References

[1] Ritchie, D. M. "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal.* Vol. 63, No. 8, October 1984, pp 1897-1910.

[2] Olander, D. J., McGrath, G. J., and R. K. Israel. "A Framework for Networking in System V," *USENIX Conference Proceedings.* Summer 1987, pp 38-45.

[3] McKenzie, A. M. *ISO Transport Protocol Specification - ISO DP 8073.* RFC905, April 1984.

[4] Postel, Jon. *Transmission Control Protocol.* ARPANET Working Group Request for Comments, Network Information Center (NIC), SRI International, Menlo Park, CA. RFC793, September, 1981.

[5] Postel, J. and J. Reynolds. *TELNET Protocol Specification.* ARPANET Working Group Request for Comments, Network Information Center (NIC), SRI International, Menlo Park, CA. RFC854, May, 1983.

[6] *UNIX System V Release 3 STREAMS Programmer's Guide.* AT&T, Issue 1, Order Code 307-227.

An Experiment with Network Shared Libraries

V. S. Sunderam

Department of Math & Computer Science
Emory University, Atlanta, GA 30322
vss@mathcs.emory.edu

Library routines, ranging from isolated formatters to complete software packages are used by virtually every program. Under several circumstances, it is beneficial and even desirable for a 'library server' on a remote machine to provide such functions to multiple clients that share this service. This paper describes the design, implementation and experience gained in the deployment of such a server on a local network of independent workstations. While the network shared library is premised on the basic RPC concept, extensions have been needed to support the preservation of state across calls, client and server failures, and server multiplexing. A network shared library for the X Window System™† has been implemented and tested; with the server on a fast processor, multiple clients have been observed to run at higher speeds with the added benefits of reduced memory and processing overheads on the client machines.

1. Introduction

Library routines, that are bound into application programs and invoked to perform a variety of tasks, are used by practically every program in existence. Libraries are used to access previously written software ranging in diversity from simple I/O formatting to mathematical subroutines to complex window system functions. The typical paradigm for the use of libraries is link edit the required object modules into the executable file of the application that wishes to use them. While the concept and use of library routines is extremely simple and efficient, there are some drawbacks. The most important disadvantages are the duplication of library routines in object files that lead to size increases and the extensive relinking (or even recompilation) efforts required when modifications are made to the library routines.

The obvious solution to the above problems is for applications to share library routines at execution time - by dynamically linking in routines from a common location as required. This is indeed an effective way of accessing library routines and several operating systems do support such schemes; [Arnold 86], [Gingell 87], and [Murphy 72] contain examples. The exact mechanism employed in shared libraries varies; static linking to well known addresses, the use of traps on the first invocation to perform dynamic linking, and virtual memory mapping to library objects are representative schemes. In most implementations, there is little or no degradation in performance and the benefits in terms of reduced primary and secondary memory as well as in ease of maintenance are great.

In recent years however, computing environments as well as applications have changed in several ways that prompt a fresh look at the use of library routines. First, local networking has become widespread, leading to the sharing of several resources by hosts (predominantly workstations) on a network. CPU cycles and secondary storage are classical examples of resources that are shared; at a higher level, software available on one machine is easily accessed from another. Second, with operating systems based on Unix™†† gaining in popularity, large software systems are being portably used on a variety of machines -

† The X Window System is a trademark of the Massachusetts Institute of Technology.

†† Unix is a registered trademark of AT & T

most often, these are made available as library packages. Examples are window systems, database systems, and mathematical software. Given the size and complexity of these software packages, the effort required to maintain them, and the fact that machines on a local network are unevenly loaded, the notion of a network shared library becomes worth investigating.

A network shared library would be capable of providing library services for a given suite of routines on a network-wide basis. In terms of reducing object code sizes and ease of maintenance, such a facility would achieve advantages offered by a shared library on a single machine, perhaps to a greater extent. The more significant advantages of a network shared library, however, are expected to be the following:

- In a heterogeneous network, a library suite that is available for only one architecture may be accessed from dissimilar machines.
- Specialized hardware support (such as a floating point accelerator) that library routines could benefit from may be available on one machine; a library server on that machine could perhaps make the performance improvements and functionality available to others in a natural and convenient manner.
- Certain machines in a network may be overloaded while others are idle. By moving the CPU cycle and memory requirements of library packages to lightly loaded machines, balancing and increased throughput are possible.
- Low cost, special purpose processing nodes on the network may act as library servers, freeing valuable resources on more general purpose machines on a network.

The viability of such a server is, of course, dependent on the cost of providing the service. On first analysis, it appears that the overheads in making procedure calls over a network will far outweigh the above benefits and degrade the performance of programs to unacceptable levels. This is certainly true for most 'traditional' library routines such as trigonometric functions and formatting procedures. However, as previously mentioned, new applications such as window systems are implemented in library routines that have completely different characteristics and requirements. Furthermore, with diskless workstations, even a local (or locally shared) library is in fact being paged over the network - a library server on a disk based system may be more efficient in such cases.

In an attempt to empirically study these ideas and to determine the conditions and circumstances under which a network shared library might be profitable, a prototype has been designed, implemented, and tested. The suite of routines chosen was the X window system library [Scheifler 86], referred to henceforth as Xlib. In this paper, the experiences gained from the prototyping experiment are reported. The next section discusses in greater detail the issues involved in library sharing across independent hosts. In the third section, an overview of the design and implementation is presented. Test results and observed measurements follow; the concluding section contains an evaluation and suggestions for future work.

2. Issues and Analysis

In analyzing the ideas behind and the viability of network shared libraries, two main aspects were studied. The first was the class of library suites that might be well suited to such an approach; the second, quantitative enumeration of the advantages of network shared libraries over conventional schemes. This section discusses these issues in detail and provides the background for the remainder of the paper.

2.1. Appropriate Library Suites

The determination as to whether a given library suite is appropriate for network sharing needs to be made on the basis of trade-offs on several dimensions. One of these is availability; when a library package is available only on a particular architecture or operating system environment, network shared libraries would allow access of functions from dissimilar machines. However, even under these circumstances, it may be more profitable to rewrite the application for the machine on which the library is available. Nevertheless, when language availability or porting efforts dictate that the client application must execute on a different machine, network shared libraries can, when the overheads are tolerable, resolve such problems. (We note here without a detailed discussion, that this circumstance in particular and network shared libraries in general may raise complex licensing issues in some situations).

Many library suites may also be beneficially implemented as network shared libraries when applications on diskless workstations wish to access them. When the routines are large but are invoked only a few times by the application, the cost of paging in the executable file may exceed that of invoking the functions over the network. For example, the **xrefresh** application in the X window system invokes five separate Xlib routines a total of ten times on each execution. The five routines and the others within Xlib that they in turn invoke, increases the object code of the application by 60 Kbytes. On a diskless workstation, assuming that all pages remain resident, paging in the Xlib routines requires approximately 150 ms. On the other hand, the ten library calls made over the network would require only 60 ms. Even though only a few applications exhibit this behavior, (increasingly common) diskless workstations could benefit substantially in several instances. Although only rough estimates could be obtained, an analysis of several applications showed that paging overheads for library routines could be up to 3 times the overheads for making the calls remotely.

Some categories of library routines operate asynchronously. In other words, some library routines initiate actions that may not have been completed even after control has been returned to the caller. Many Xlib routines fall into this category. As Figure 1 shows, the Xlib routines communicate asynchronously with the X server; a client request to a line drawing library routine may return well before the X server has actually performed this operation. The implication in such instances is that the application does not require that a routine complete all of its actions in order to proceed with its own execution. The traditional notion of a procedure call is no longer strictly valid; however, the IPC and asynchronous aspects are invisible to the application which may continue to use the procedure call model. With window systems, database servers, and various toolkits increasingly utilizing such schemes, partially asynchronous library suites will become common. When immediate completion of library actions is not critical, network shared libraries can help in offloading CPU and memory requirements at no expense to the application - while optimizing system wide resource usage.

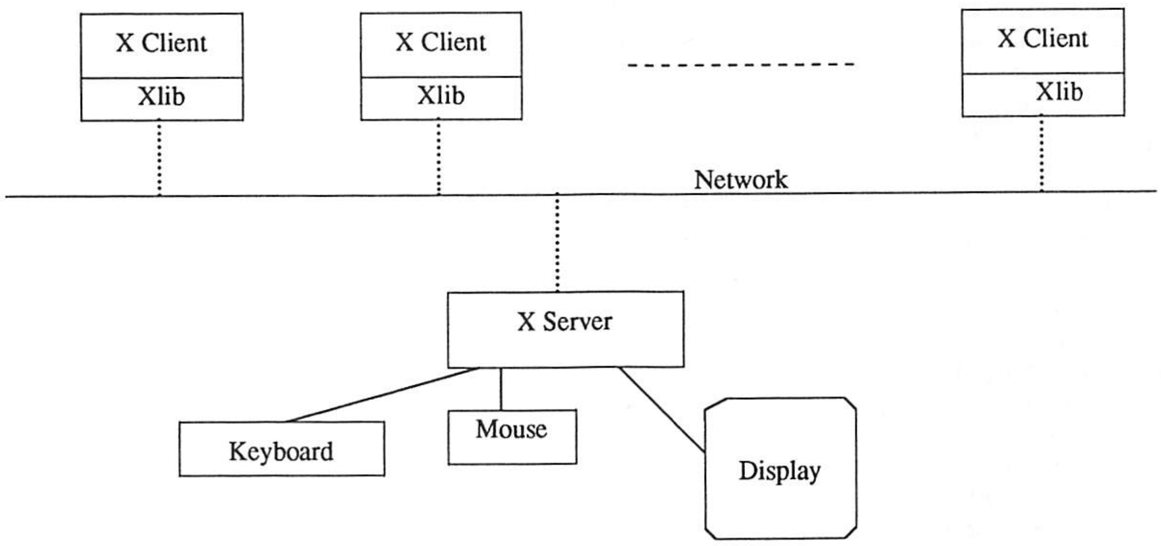


Figure 1: X Windows System Structure

The above discussion attempted to identify properties of library suites that make them suitable for network sharing. Similarly, certain suites, by their nature, are unsuitable for remote access. The standard mathematical library routines, for instance, are small (a few hundred bytes each), short running, and return results that must be available for the caller to continue execution. Implementing these as a network shared library would incur prohibitive overheads and would not be beneficial. Certain other library suites, such as I/O routines, refer to underlying objects that are only meaningful within a single host processor, thereby precluding their execution on remote machines. It should be noted that both these cases are classic examples of traditional library suites - as mentioned, trends in newer software systems and computing environments pose a different set of characteristics that reinforce the arguments for network shared libraries.

2.2. Parameter Passing

When conventional library routines are bound into an application's executable file, both the caller and callee normally assume that they will execute in the same address space. As a result, call-by-reference parameters (and sometimes even access to global variables) pose no problems. This is true in most implementations of local shared libraries also. As a result, library routine invocation and most aspects of the execution are exactly the same as with calls to procedures within the application itself.

In the case of network shared libraries, owing to the separation in address spaces of the caller and the library routines, special provisions (such as "call by copy/restore") have to be made to handle pointer parameters and globals. Most RPC systems prohibit the use of such parameters [Tanenbaum 88] and such parameters can be significant obstacles in network shared libraries. Often however, in complex library suites, it is observed that some data structures are never accessed by the application although pointers to these structures are passed to the library routines as parameters. An example is the 'Display' structure in Xlib that contains information concerning the graphics display device. An Xlib routine constructs this structure and returns a pointer to it to the application; structure elements are accessed and modified only by other Xlib routines although the pointer is a parameter in virtually every call made by the application. When such situations are identifiable, these data structures may simply be allocated in the address space of the library server and the pointers alone passed as parameters without the need for copy/restore. In a later section, general guidelines for resolving this issue and specific schemes adopted in this project will be discussed.

2.3. Library and Application Failure Handling

In situations where the functions of an application are distributed among independent processes, partial failures are possible, leading to inconsistencies and undesirable actions in the remaining components. In the case of RPC, client failures (hardware or software) lead to 'orphan procedures' in the server; a server failure may cause the client to be suspended indefinitely. Traditional RPC schemes detect server failures using timeouts and prevent orphans by discarding results from procedures when a client fails after procedure invocation. Several implementations of RPC have attempted to build in recovery or fault-tolerance schemes; concentrating primarily on server failures. Representative examples may be found in [Yap 88] and [Aschmann 87]. Most of the methods employed involve replicated execution of server procedures. In the case of network shared libraries, the issues in failure detection and recovery are somewhat more complicated owing to the fact that state information may be maintained in the server.

As described in the previous section, application data structures may be allocated in the server's address space and referred to by clients that use opaque pointers to them. If fault-tolerance to server failures is to be improved, replicated server execution alone is insufficient since data structures in different server instances may be allocated at different addresses. Server replication schemes must therefore ensure that instance dependent values are not used by clients as opaque handles; not doing so, however, is bound to increase overheads considerably. Another serious drawback is caused by the likelihood of network library calls being non-idempotent. In such instances, multiple execution of server procedures will lead to incorrect results. In view of these factors, recovery by using checkpoint restart schemes (e.g. [Mandelberg 87]) appears to be the most viable solution and are being investigated. In the case of client failures also, it is no longer sufficient to simply discard results from orphan procedures. To avoid forms of deadlock and the accumulation of resources, data structures pertaining to a failed client must be released in a graceful manner. Therefore, servers must be able to explicitly recognize client failures as well as possess knowledge regarding their resources and the correct manner in which to free them. Work is in progress to incorporate synchronization features into the communication protocol to enable clients and servers to 'rollback' to a consistent state upon restart after failures.

2.4. The X Library Suite

The X window system has gained tremendous popularity over the last few years and is the window system of choice on many platforms. One of the most interesting aspects of X is the fact that it is network based; the "server" that interfaces to the display, keyboard, and mouse communicates with "client" applications over a network connection. Client applications interact with X using procedure calls; the X library suite contains well over 300 routines for various window system functions. In the absence of shared libraries, Xlib routines that are directly or indirectly referenced are bound into the application object code.

Given the complex nature of the library functions and their numbers, it is not surprising that the executable files of typical X clients contain upwards of 70 kbytes of library routine code. In many cases, this makes up over 50% of the size of the object files. When workstations execute many X clients simultaneously, swap space and memory requirements are increased enormously and could lead to degraded performance. Another disadvantage of Xlib and other large library suites is the increase in disk space requirements; an Xlib network shared library was therefore viewed as a potential solution to these problems. Xlib is in fact an ideal candidate for network shared libraries for many reasons including size, widespread use in workstation network environments, software maintenance factors, and, most importantly, its inherent asynchronous operation.

One of the difficulties in realizing an existing library suite as a network shared library is that of distinguishing between input, output, and input/output parameters to procedures. In RPC programming, the programmer has explicit knowledge of these and other factors such as side-effects; this knowledge is necessary for the correct generation of RPC stubs. In converting existing applications and libraries however, detailed program analysis is necessary to determine these issues and generate appropriate stubs. Methods for such analyses exist (e.g. [Allen 74], [Barth 78]) and are being investigated as the basis for potentially automated stub generation schemes. In the case of Xlib, however, a rigid naming convention for Xlib function parameters permits the straightforward separation of most parameters into input and output classes. As previously mentioned, many of the data structures used may be allocated within the Xlib server and referenced by clients using opaque pointers; only a few structures that are required to be visible to applications and the library server need to be copied in their entirety on procedure calls.

3. Design and Implementation

The primary objective of this project was to determine the feasibility and value of network library servers by experimenting with a widely used software system, namely the X window system. To this end, an Xlib server was constructed and representative X clients rebuilt to use this Xlib server. This section describes the salient features of the design and implementation; the following section reports on actual observations in the use of the network based Xlib.

3.1. Operational Overview

The X window system consists of client applications that make requests to the server to perform various window system functions. Client requests are always made by library routines on behalf of the applications; traditionally, the required libraries are bound into the application object file. In the case of the network based Xlib, the library server is an independent process that accepts requests from applications and forwards them to the server, returning appropriate result values to the caller. The Xlib server may execute on any host on a local network and may service requests from multiple applications. The Xlib server may interact with multiple X servers depending upon the requirements of the application program(s). The existence of a network based library server is completely transparent to the applications and they require no modifications; however, in the case of Xlib, recompilation and the substitution of one header file was necessary as will be explained in the following subsection. With a network based Xlib, an application's library procedure calls are transformed to remote procedure calls with added facilities to mask the remote nature of the library server and to detect and recover from errors. Figure 2 depicts the network based library server as contrasted to local libraries.

3.2. Client Interface

The X library suite consists of several hundred routines that are invoked using procedure calls by X clients. In order to permit these calls to be executed remotely without necessitating changes to client programs, a special header file is provided that contains a macro definition for each Xlib function. The macros expand into RPC stubs that contain statements for packing the function arguments, invoking the remote procedure, unpacking the returned arguments and returning the results. In the interest of portability, these statements themselves are macros that may have different expansions in different environments. The special header file contains all the stub macro definitions as well as a few additional variables used by the stubs. In addition, the header file also contains macros that generate remote procedure calls for a small set of built-in X macros. The built-in X macros obtain information regarding objects that are assumed to exist in the address space of the application; in the network based Xlib, these assumptions are invalid necessitating their implementation as remote procedure calls. The least disruptive method to do this was to replace the header file "Xlib.h" by the special header file.

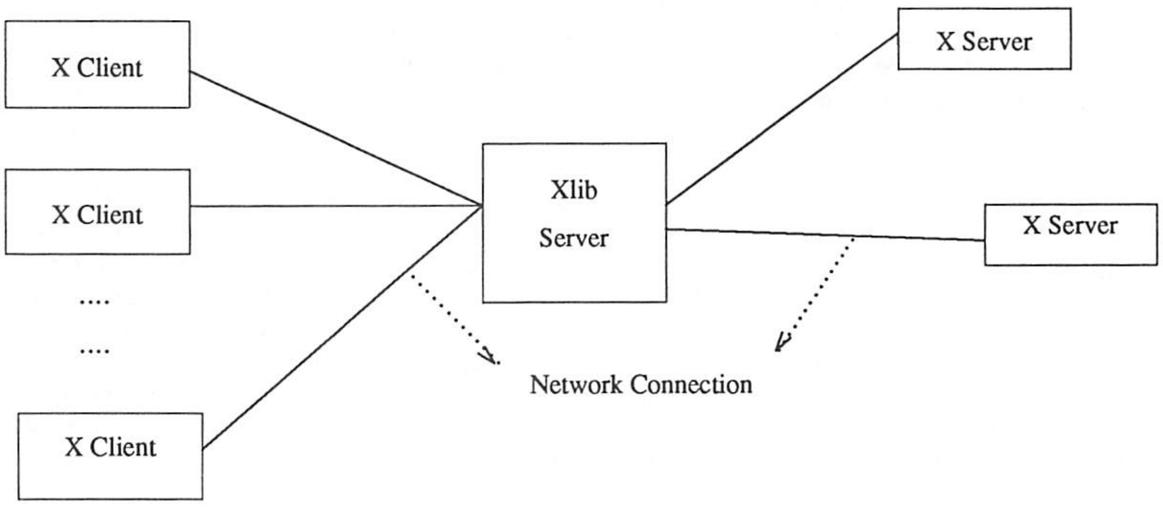


Figure 2: Network Shared Xlib Structure

The X library suite follows a rigid naming convention for the parameters to its routines. In particular, the suffix '*return*' is applied to parameters that contain return values. This convention greatly simplified the stub generation process, most of which was done automatically. A simple stub generator was developed that analyzed the header for each Xlib function and output the macros necessary to perform a call to that particular function remotely. Simple data type parameters could be identified easily, as could variables used to return results. Those data structures that could be opaque to the client were pre-identified manually; the stub generator treated such parameters as typed pointers without regard to the structure components. A few data types generated macros to copy and transfer the entire structure. The only areas of inconvenience were in cases where dynamic memory allocation was required and routines to which function names were parameters; the few instances of these were handled manually. Each stub also contained macros to receive and interpret error results from the Xlib server.

When the Xlib functions are remotely accessed by clients, failures in the Xlib server do not automatically terminate the client. The 'correct' action to be taken in such situations is debatable; requiring the client to abort and an attempt to use a different server are both reasonable approaches. The latter solution would permit the client to continue execution if the Xlib server could be restarted or migrated without loss of state information. Several migration and restart systems exist; however, none of these could be directly employed without modifications to the X server. A facility that would permit the Xlib server to continue or resume execution without changes to the X system is being investigated. At present, the client process is terminated in a manner identical to the situation when the Xlib routines are linked into the client or shared locally.

3.3. The Xlib Server

The Xlib server executes as an independent process on any host on a local network. Multiple Xlib servers may be simultaneously operational; in the test implementation, they are manually initiated. Clients determine the location of a server by broadcasting a local network level request to which all active servers respond indicating the number of clients they are currently serving. The client selects the least loaded server from among those that respond within a timeout period and connects to it. This decision is also influenced by information regarding the machine architecture on which the server is executing; similar hosts are preferred to avoid the need to transmit requests in host independent form. Clients and servers determine this at initial connection time and use XDR[Sun 87] encoding only when necessary.

Xlib servers consist of a small dispatch program to which all Xlib routines are bound. The dispatch code was generated in a manner similar to that of creating client stubs. This code contains macros to unpack the arguments and invoke the appropriate Xlib routine. Upon return, the results are packed and returned to the caller. In addition, when data structures are dynamically allocated in the Xlib server a record of these is maintained for each client. This permits the deallocation of such structures if a client were to terminate abnormally.

The Xlib server is a normal Unix process and is therefore single-threaded. Calls from multiple clients are serviced in sequence; multiplexing between clients is straightforward and efficient if **poll** or **select** mechanisms are available. Most Xlib routines are output oriented in that they cause the X server to perform a graphics related function. As the communication between the X server and the Xlib routines is asynchronous, there is no problem with respect to the multiplexed, sequential handling of calls to such routines. However, certain Xlib routines block the caller until the X server notifies it of an event occurrence. Xlib routines that exhibit this behavior are not invoked by the dispatch program until input is available on the Xlib-Xserver connection.

4. Experiences and Results

In order to obtain quantitative evaluations of the benefits of using network shared libraries, the X library suite was implemented as a network server. Several standard clients were recompiled to make use of the Xlib server. The cross-section of clients chosen included long running, low activity applications such as the load average monitor (**xload**), moderate input rate programs such as **xterm**, high output rate demonstrations (**ico**), and miscellaneous clients (e.g. **xcalc**). In addition, a benchmarking program (**gbench**) was employed in order to obtain performance data. This section presents the results obtained by experimenting with the Xlib network shared library.

4.1. Performance Issues

As mentioned earlier, one of the foremost concerns in implementing library software on network based servers is the possible degradation in execution speeds of the applications. In order to determine the impact on performance, various combinations of clients in terms of type and number were run simultaneously against Xlib servers on remote machines. In these trials, visual perception was used to determine if any degradation could be observed; quantitative results are presented later in this subsection. The overall impression was that depending upon the Xlib server machine and the mix of clients, performance was equal to or better than statically linked clients for between 4 and 14 clients per server. Table 1 indicates representative examples of the mix and number of clients at which performance was noticeably affected for various combinations of client and server machines. The term 'noticeable degradation' is, of course, subjective - comparatively jerky output, echo delays, and lowered response were used as criteria. However, more precise quantitative measurements were also obtained using the **gbench** graphics benchmarking program and are described in the following section.

Client Machine (all clients)	Server Machine (single Xlib server)	Noticeable degradation with: (client mix)
Sun 3/50	Sun 3/60	(a) 6 (xterm,xload,ico) (b) 5 (xterm,2*ico) (c) 8 (xterm,xclock,xload)
sun 3/60	Sun 3/60	(a) 5(xterm,xclock,ico) (b) 4(xterm,2*ico) (c) 6(xterm,xcalc,xload,xclock)
Sun 3/60	Sun 4/280	(a) 9(xterm,xclock,2*ico) (b) 12(xterm,xclock) (c) 14(xterm,xload,xcalc)

Table 1: Load handling capacity of Xlib server

The **gbench** program is a graphics benchmarking utility that measures the maximum rate at which graphics functions can be performed. This program was used to obtain a comparison of client performance in quantitative terms with the Xlib routines statically linked and with a remote Xlib server. Table 2 contains the observed performance figures for a variety of graphics functions. This table shows data for a single **gbench** process; in most cases (i.e except the trivial point drawing operation) the rate at which graphics functions could be performed are only slightly lower with the network based Xlib. This lowered performance is naturally due to network overheads, but it is worth noting that like most benchmarks, **gbench** repeatedly performs the same operations in rapid succession with little intermediate processing - a behavior which real applications rarely exhibit. However, when multiple gbench processes are simultaneously executed, a remote Xlib server on a fast processor actually increases the rate at which graphics operations are done, as shown in Table 3.

Operation	Local X library	Sun3/60 Xlib server	Sun4/280 Xlib server
Arc drawing	3.7/sec	3.4/sec	3.7/sec
Blit drawing	72.1/sec	54.6/sec	69.3/sec
Polygon	9.4/sec	9.4/sec	9.4/sec
Filled Polygon	102.4/sec	81.3/sec	94.1/sec
Rectangle	10.0/sec	10.2/sec	10.7/sec
Text	185.4/sec	122.0/sec	125.5/sec
Point	6093.2/sec	211.3/sec	289.4/sec
Vector	71.8/sec	58.8/sec	68.2/sec

Table 2: Comparative speeds of **gbench** operations

Operation	Local X library	Sun3/60 Xlib server	Sun4/280 Xlib server
Arc drawing	2.6/sec	2.4/sec	3.5/sec
Blit drawing	44.1/sec	35.6/sec	59.3/sec
Polygon	8.2/sec	7.4/sec	9.0/sec
Filled Polygon	61.4/sec	47.3/sec	74.1/sec
Rectangle	7.1/sec	6.2/sec	8.7/sec
Text	145.4/sec	103.0/sec	125.5/sec
Point	4122.2/sec	181.3/sec	283.4/sec
Vector	47.4/sec	38.8/sec	58.2/sec

Table 3: Average speeds with multiple (3) **gbench** clients on one host

To summarize, these experiments indicate that a network based Xlib server performs adequately and there is no drastic degradation in performance. Although the limit on the number of simultaneous clients per server seems small, it should be mentioned that these experiments attempted to induce maximum client activity and include hyperactive applications like the **ico** demonstration program. In most practical situations several clients are intermittently idle and only invoke Xlib functions periodically, permitting a larger number of clients to use a single Xlib server. Table 4 shows the typical X server request patterns for some common clients; only a small number of requests per second, each of small size, are made by most applications in normal use. Furthermore, the other advantages of network shared libraries may compensate for any failing in performance.

4.2. Object Sizes

One of the motivations behind network shared libraries is the anticipated benefits resulting from reduced sizes of application accessing the library routines. This reduction can be very substantial and lead to reduced secondary storage requirements as well as lower paging overheads when the applications execute on diskless workstations. To obtain a measure of the actual differences, several X clients were compiled twice; in one case using statically linked Xlib routines and in the other, a network shared Xlib server.

Table 5 shows the reduction in sizes of the object code files. In absolute terms, object files become smaller by between 70 and 150 kbytes which equates to between 30 and 50% of the total size of the executables.

Application	Client->Server (avg. bytes/request)	Server->Client (avg. bytes/response)	Frequency
ico	172	32	6-8 times/sec
xterm (user input)	60	32	2-7 times/sec
xterm (cat termcap)	2000	32	1 time/sec
xload	20	(none)	Update frequency
xclock	128	(none)	Update freq. (1-60 secs)

Table 4: Request patterns for common X clients

Application	Statically linked	Network based Xlib
xterm	254k	128k
xclock	212	112k
xcalc	192k	96k
xload	140k	64k
gbench	286k	196k

Table 5: Comparative sizes of X clients

The figures shown in Table 5 are somewhat outdated in the sense that they pertain to applications that used only the low level X library routines or 'intrinsics'. In recent releases, the X window system includes 'toolkit's that add an intermediate layer between the application and Xlib for the purpose of simplifying X client development. The toolkit related functions have not yet been implemented as a network shared library, but some data on the current situation is presented below. Table 6 lists the sizes for X clients from the latest release (V11R3) for the statically linked case and when the Xlib (intrinsics) alone are implemented as a network shared library. It is observed that the size reduction is not as dramatic - they range from 45 to 70 kbytes. With the introduction of toolkits and its ubiquitous use, however, a natural progression is to implement the toolkit functions themselves as a network shared library. The next phase of this project will incorporate the Xt and Xaw routines into the network based Xlib server; Table 7 shows the expected effect based on estimated client sizes.

Application	Statically linked	Network based Xlib
xterm	352k	290k
xclock	245k	176k
xcalc	140k	96k
xload	222k	160k

Table 6: Comparative sizes of X clients (X11 Release 3)

Application	Statically linked	Network based Xlib
xterm	352k	192k
xclock	245k	96k
xcalc	140k	96k
xload	222k	96k

Table 7: Size comparisons if Xaw, Xt, Xlib are network based

5. Conclusions and Future Work

This project investigated the feasibility of remotely sharing library routines on a local network by building a prototype network shared library. Like many experimental projects, a few aspects of the design and development are somewhat ad-hoc in nature and the criteria for evaluation is partially subjective. Nevertheless, it is clear that network shared libraries are viable and provide benefits along several dimensions under many circumstances. section attempts to analyze the value of network shared libraries and outlines directions for future work.

5.1. Locally Shared Libraries

Libraries that are shared *within* a host environment appear to provide most of the advantages that network shared libraries do and, in fact, may prove to be a more effective solution in certain circumstances. However, there are several compelling reasons for the remote implementation of shared libraries - some less tangible than others. Availability is a significant factor; network shared libraries provide a means for the access of library functions available only on a particular remote host. A second consideration is maintenance and control efforts - network shared libraries can be updated independently of the client applications, fewer copies need to exist, and multiple versions may exist simultaneously. From the load balancing point of view, network shared libraries potentially enable the distribution of computations across hosts in a network. Performance improvements are also obtained, particularly when clients execute on diskless workstations (with relatively little physical memory) - this factor is highlighted further by the ever-increasing sizes of typical library packages.

5.2. Conversion Issues

One of the major obstacles in the use of network shared libraries is the process of converting applications and library routines. Replacing routine invocations by equivalent remote calls is, in general, not possible without detailed analysis to determine the nature of parameters, the extent of data sharing, and to identify data structures that can be allocated in the library server. In the case of Xlib, rigid naming conventions and manual analysis resolved this issue in a straightforward manner but other library suites may preclude such a solution. It is planned to study side effect analysis techniques such as those described in [Allen 74] and [Barth 78] in order to evolve an algorithmic method for the conversion of applications and for the generation of server dispatch programs.

5.3. Performance

As demonstrated by the described experiments, the Xlib network shared library performs well and does not introduce significant overheads. However, one of the factors contributing to good relative performance was the already asynchronous nature of X server - client communication. Other library suites must be tested before general conclusions can be drawn. Another factor to be considered is that all the applications tested belong to the standard or 'core' X distribution in a development environment; production applications in the domains of engineering design or high-performance graphics may exhibit different performance characteristics. Work in progress in this regard includes the development of high throughput protocols (e.g. [Sunderam 88]) for use between clients and the Xlib server. The possibility of using these protocols for Xlib - X server communications is also being considered, as this will permit the migration or restart of the Xlib server in case of failures.

5.4. Space Considerations

The static analysis of object code sizes described in the previous section indicates that substantial savings in space can be obtained by using network shared library servers. Of course, similar saving can be obtained with local shared libraries; however, not all operating systems support such a mechanism whereas network shared libraries may be implemented above existing facilities. Furthermore, availability considerations as well as factors such as physical memory and computational power on the client machines must be taken into account. Preliminary tests show that on a Sun 3/60 with 4 Mbytes of memory, nearly twice the number of X clients (a mix consisting of demonstrations, terminal windows, and miscellaneous utilities) could be run with equivalent performance when using a network based Xlib server as opposed to local shared libraries. The Xlib server in this experiment executed on a Sun 4/280 with 32 Mbytes of memory but its effect on the server machine was negligible. The next phase of this project will attempt to obtain accurate measurements of processor load, CPU and memory utilization, and paging activity to quantify the benefits of network based library servers.

6. Acknowledgements

The author is indebted to the referees for their valuable comments and suggestions.

7. References

[Allen 74] F. E. Allen, "Interprocedural Data Flow Analysis", *Proceedings of the 1974 IFIPS Congress, 1974.*

[Arnold 87] J. Q. Arnold, "Shared Libraries in Unix System V", *Proceedings of the Summer 1986 Usenix Conference*, June 1986.

[Aschmann 87] H. R. Aschmann, "Resilient Remote Procedure Call", in *Distributed Processing (Barton et. al. eds)*, IFIP, North-Holland, 1987..

[Barth 78] J. M. Barth, "A Practical Interprocedural Data Flow Analysis Algorithm", *Communications of the ACM*, Vol. 21, No. 9, September 1978.

[Gingell 87] R. A. Gingell, M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS", *Proceedings of the Summer 1987 Usenix Conference*, June 1987.

[Mandleberg 87] K. I. Mandelberg, V. S. Sunderam, "Process Migration in Unix Networks", *Proceedings of the 1988 Winter Usenix Conference*, February 1988.

[Murphy 72] D. L. Murphy, "Storage Organization & Management in TENEX", Proceedings of the FJCC, AFIPS, 1972. No. 5, November 1987.

[Scheifler 86] R. W. Scheifler, J. Gettys, "The X Window System", *ACM Transactions on Graphics*, Vol. 5, No. 2, April 1986.

[Sun 87] Sun Microsystems, Inc., "XDR: External Data Representation", *Internet Request for Comments : RFC1014*, June 1987.

[Sunderam 88] V. S. Sunderam, "A Fast Transaction Oriented Protocol for Distributed Applications", *Proceedings of the 1988 Winter Usenix Conference*, February 1988.

[Tanenbaum 88] A. S. Tanenbaum, "Computer Networks", Prentice Hall, 1988.

[Yap 88] K. S. Yap, P. Jalote, S. Tripathi, "Fault Tolerant Remote Procedure Call", *Proceedings of the 8th Intl. Conf. on Dist. Computing Systems*, June 1988.

The Mether System: Distributed Shared Memory for SunOS 4.0

Ronald G. Minnich
Supercomputing Research Center[*]
Bowie, Maryland
and
David J. Farber
Department of Computing and Information Sciences
University of Pennsylvania
Philadelphia, Pennsylvania

Abstract

Mether is a Distributed Shared Memory (DSM) that runs on Sun[1] workstations under the SunOS 4.0 operating system. User programs access the Mether address space in a way indistinguishable from other memory. Mether was inspired by the MemNet DSM, but unlike MemNet Mether consists of software communicating over a conventional Ethernet. The kernel part of Mether actually does no data transmission over the network. Data transmission is accomplished by a user-level server. The kernel driver has no preference for a server, and indeed does not know that servers exist. The kernel driver has been made very safe, and in fact *panic* is not in its dictionary.

The Mether system supports a distributed shared memory. It is distributed in the sense that the pages of memory are not all at one workstation, but rather move around the network in a demand-paged fashion. It is shared in the sense that processes through the network share read, write, and execute access. And it is memory in the sense that user programs access the data in a way indistinguishable from other memory. The memory is never paged to disk, but the delay of accessing a page over the network is approximately the same as a paging disk.

Two examples of Mether programs are shown in Figures 1 and 2. Note that, aside from the call to **methersetup** these programs look quite ordinary. One program prints out the value of the first 278 bytes of Mether memory; the other clears the first page of the Mether memory and then increments each byte 128 times. If the first program is running the values displayed increase. You can run either program on any host that supports Mether. The writer takes about 8 seconds to run, whether the watcher is running or not. In fact the writer usually runs a little faster if the watcher is on another machine.

As the examples show, programs that access this memory can pretend that it is normal memory. If they do they may pay a substantial performance penalty. As shown in [4] programs that use DSM without modification rarely show the sort of performance gain found on a conventional shared-memory multiprocessor. Programs must be more careful; if they are then they can communicate across the network at apparent memory speeds.

The memory is accessed by opening a special file. Once the file is opened the user program executes an *mmap* system call and maps the area into its address space. From that point on the process may treat the memory as it would any other memory. A function library is provided to make the use of Mether totally transparent.

[*]This work was done while the author was at University of Delaware, Newark, De.
[1]Sun and SunOS are trademarks of Sun Microsystems Inc.

```
#include "world.h"
main()
{
  unsigned int i, j;
  initscr();
  methersetup();
  while(1)
    {
      move(0,0);
      for(i = 0; i < 0x120; i += 0x10)
        {
          printw("%08x:", METHERBASE+i);
          for(j = i; j < i + 0x10; j++)
            printw("%x ", * (unsigned char *) (METHERBASE + j));
          printw("\n");
        }
      refresh();
      sleep(1);
    }
}
```

Figure 1: The Mether watch program.

```
#include "world.h"
main()
{
  int i;
  unsigned char *p = (unsigned char *) METHERBASE;
  methersetup();
  for(i = 0; i < 8192; i++)
    *p++ = 0;
  for(p = (unsigned char *) METHERBASE; *p < 0x80;
      p = (unsigned char *) METHERBASE)
    {
      for(i = 0; i < 8192; i++, p++)
        *p += 1;
    }
}
```

Figure 2: A program that writes to an Mether page

If the process is the only one using an area of the memory, then it will run at full memory speed. If other processes on the same processor are using the same area, they will all run at full speed, unless one of the other processes locks an area of the shared memory. If processes on other processors simply read the memory infrequently there will be a small impact on writes as messages are sent out to the other processors invalidating their copy (or, in the current protocol, updating their copy). If many processors write the same location frequently then there will be a substantial performance degradation, probably only allowing a few thousand operations per second. Mether is non-blocking so the processor will not be slowed down, just the processes accessing the contended-for location.

Mether is inspired by a high-speed memory-mapped network built at the University of Delaware by Delp and Farber. We give a cursory description of MemNet below; for more details see [1], [2] and [3].

MemNet is a memory-mapped network. MemNet provides the user with (in the current implementation) a two Mb contiguous region of memory which is shared between a set of processors. The sharing is accomplished using dedicated page-managment hardware communicating via a high-speed token ring. When a MemNet page is needed and it is not present in the local interface a message is sent over the token ring requesting the page. The hardware provides consistency between pages. The algorithm used is similar to those used for snooping caches: when a chunk is written all other copies of that chunk are invalidated before the write completes. For performance reasons the pages are only 32 bytes long. This size was decided upon as the optimal tradeoff between transmission time and several other factors. For a complete performance analysis, see [1].

On a system such as MemNet the global address space is much larger than any single interface's memory. A problem that must be addressed is what to do in the event a chunk can not find an interface with room for it. Some interface must always keep the space open for that particular chunk (address) in the MemNet address space. To address this problem MemNet supports the notion of *reserved memory*. Reserved memory is the set of chunks for which a particular interface is responsible. Space will always be available for these chunks in the interfaces' reserved area. If no space can be found for a chunk on any interface in a non-reserved area, the chunk will end up back in the reserved memory in the interface which is its home. If MemNet did not support reserved memory, chunks might be lost as interfaces filled up with multiple copies of chunks. In general a MemNet interface will have a "fair share" (i.e. on a system with 10 interfaces, 10%) of its memory as reserved memory, with the rest of the memory available for other chunks.

Mether supports reserved memory too, on a page basis. In fact, a page must be in the reserved memory of some Mether interface for it to be created. In other words, pages are created only from the reserved space, and only when they are referenced. When a non-reserved page is referenced for the first time on a processor, a request for that page is sent out. Only if that page is in some processor's reserved address space will space for it be allocated.

One difference between MemNet and Mether is that Mether blocks the process when a page is unavailable whereas MemNet blocks the processor. This difference is more important than might at first seem. On MemNet, hot spots can consume the process, the network, and all the processors on the network. It is essential that algorithms be well-behaved. Otherwise the processors on the network can, in the absolute worst case, run orders of magnitude slower than normal. On Mether only the processes requesting the information are affected. Other processes, processors, and the network operate normally.

We wanted to gain experience with a DSM that ran on more than the three processors available on the existing MemNet network. Our goal is to build a DSM that matches MemNets' best-case and worst-case performance. In the best case, MemNet runs at memory speeds; in the worst case, it is several orders of magnitude slower. One reason that Mether makes no attempt to minimize paging latency is that we want to get as close to the MemNet environment as possible and explore ways in which to use that environment correctly.

We will describe Mether in further detail below, after which we will describe factors that constrained the design. Mether is driven by MemNet-inspired constraints; there were a number of other constraints, driven by both technical and political realities.

1 Description

Mether provides an 4 Mb address space which is accessed via the mmap(2) system call. A typical application will open a special device using the Mether library. The *methersetup* function in the library opens the raw Mether device and performs an mmap for the entire Mether address space. As a result of this mmap the kernel does all the housekeeping necessary to support a segment comprising some or all of the Mether address space. Note that on SunOS 4.0 the first mmap is for housekeeping only, and the real mmap does not occur until the process accesses the memory. Therefore we can afford to do the initial mmap; it consumes no Mether resources.

When a page is not present on the local machine the user-level server will request it from the machine currently owning it. Currently the user level server does not support shared read-only pages that span machine boundaries. This enhancement is being implemented.

Once the page is mapped in it may be mapped out again for a number of reasons. To keep a page from being mapped out the process may lock it. While a page is locked no other process may access it. A process may choose to block while waiting for a page to be locked, or it may issue a non-blocking lock request. A blocking lock request will proceed only if the page is present on the local machine and is not locked by someone else. Otherwise the process is blocked until the page is available. Lock requests for a page not present on the local machine will cause a request to be sent out on the network for the page.

The page may also be paged out (i.e. not available on the local machine, and assumed to be present on some other machine). If the page is paged out the mmap call will sleep until the page becomes available.

Up to this point we have discussed Mether mostly in terms of the user's environment. The user calls methersetup and from then on sees ordinary memory, whether the user's processes span machine boundaries or not. In implementation Mether is divided into a kernel driver and a user program. The kernel driver manages the in-memory pages, and the user program manages the transport of pages over the network. There is nothing special about the user program that distinguishes it to the kernel driver. We describe the kernel driver and the user program below.

1.1 Kernel Driver

The kernel driver is responsible for maintaining a set of pages and their associated state. Control of the kernel driver is accomplished using the *ioctl* system call. A table of the ioctl calls and their function is shown in Figure 3.

Initially, only one type of ioctl is supported, the one which initializes the driver. Currently the size of the Mether address space is fixed at 8 Mbytes, and the virtual address range starts at `0x400000`. The parameters which must be set are the base and the size of the reserved address range for the driver. This ioctl is called METHER_INIT. The parameters passed are the start and end of the reserved address space.

Once the driver is initialized, other requests are honored. Until it is initialized all other requests return EINPROGRESS.

There are two ways to lock a page. One is METHER_LOCK, which is a blocking request for a lock. If the page is unavailable due to being locked or paged out the process sleeps. The other way to lock is
METHER_LOCKNOBLOCK, which will return either the error EAGAIN or no error, but which will not block. For both of these if the page is paged out it is marked as wanted. The user level server will attempt to fetch "wanted" pages from other processors.

Name (METHER_)	Parameter	Function
INIT	mether_init *	Initialize the Mether driver. Set the Reserved range.
LOCK	u_long *	Lock a page. The key the driver uses is not the PID, but the minor device number, for reasons explained in the text. If the page is locked or is paged out, the wanted bit is set in the page's status structure. The process sleeps until the page is available. All locks are removed when a process exits if it is that last process holding the minor device open.
LOCKNOBLOCK	u_long *	Lock a page. The same semantics as METHER_LOCK, but if the page can not be locked the driver returns EAGAIN (try again). The wanted bit will be set anyway. This can be used to implement pre-fetching of need pages.
UNLOCK	u_long *	Unlock the page. All sleepers on this address are awakened.
PAGEOUT	u_long *	The page is marked paged out. Paged-out status is not cleared when the marker exits. METHER_LOCK requests and mmap request will sleep on paged out pages, and in addition the page will be marked wanted.
PAGEIN	u_long *	The paged out bit is cleared. The wanted bit is cleared as well. The page is locked; the process doing the pagein must unlock it to make it available to other processes.
FREEALL	u_long *	All minor devices are summarily closed (including the user-level server!) and all pages are freed. The driver will once again honor only METHER_INIT requests.

Figure 3: The *ioctl* calls for the Mether device

The driver keeps track of who locked the page. The identifier used is not the traditional process id but instead the minor device number. This is done so that parents and children can share a minor device number and hence locks. This is not a sophisticated mechanism but has the virtue of being simple, fast, and taking advantage of the way child processes inherit files.

One rule guiding the use of DSM is that if you need a page when you ask for it it is already too late; you are probably going to suffer network latency. Some form of pre-fetching is desirable, and in fact was considered for MemNet hardware. Users of Mether can accomplish pre-fetching in two ways:

- perform the METHER_LOCKNOBLOCK ioctl. If the page is on some other processor it will be marked "wanted" and the user server will fetch it. This is a polling sort of pre-fetch. The process repeatedly performs the LOCKNOBLOCK until it succeeds. Until it succeeds the process may do other work.
- Spawn a child, and have the child perform METHER_LOCK. The child can signal the parent when the page is present. This pre-fetch may be more efficient since there is no polling and no chance of missing the page between polls. It does require more programming at the user level, however.

To unlock a page one uses the METHER_UNLOCK ioctl. All locks are cleared when the open count of the minor device goes to zero (i.e. all potential users of the lock release it). Note that multiple processes accessing a lock need to be careful in this environment because processes that lock a page and exit may leave it locked if child or parent processes still have the minor device open.

A more permanent way of making a page inaccessible is to page it out. A page marked "page out" is not available locally but present on some other machine. Pages acquire the paged out status in one of two ways:

- When the driver is initialized, all non-reserved pages are marked paged out.
- Any user-level program may mark a page as paged-out. Typically only the user-level server does this.

In a sense paged-out is a misnomer; the process marking the page can continue to access it. Indeed, it must if it is to copy it out and send it over the network. The name is an indication of the proper use of the ioctl, however: it should only be used by processes intending to send the page over the network. Mmap calls will block on a paged-out page, as will METHER_LOCK; these calls will also mark the page as wanted.

To page in a page the process uses the METHER_PAGEIN ioctl. Paging in a page will cause it to be locked; the data may be copied in and the page unlocked for use by other processes.

1.1.1 Using Mmap

To access the Mether address space under SunOS 4.0 a segment must be built. The SunOS 4.0 memory management scheme differs radically from other Unix[2] systems in that it is composed of paged segments. Each segment has an address range and fault handlers. For the Mether device the fault handler[3] will invoke the Mether mmap function to get a kernel page frame number. The upshot of all this is that the Mether mmap function gets called at least two times for each page: once when the segment is being built, to see when an address is valid; and once for each time the page is referenced and is not valid. The first "probe" mmap is distinguished by all the protection bits being turned on. Data for a page is never allocated until the first non-probe mmap for that page. If the page is locked by another process or paged out, the mmap will block. Once the page is available the driver makes sure that a physical page has been allocated for use and returns.

[2]Unix is a trademark of AT&T Bell Labs

[3]For those of you familiar with 4.0 Mether works as a segdev

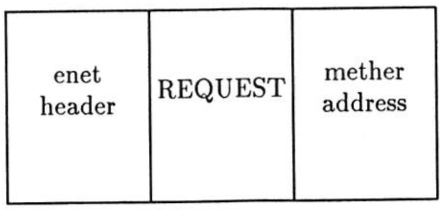


Figure 4: The request packet format

The page may be mapped out of a process's address space at any time, unless the process has locked it. Most often a page will be mapped out when the user level server needs to send it out on the network. Less often some process on the same processor must lock it, usually to prevent it from being paged out. Too much locking is anti-social and may indicate a badly designed algorithm.

When a process locks a page the kernel driver must unmap the page from all other processes that have it mapped in. The Mether device finds the Mether segment in a processes address space and unloads [4] the hardware translation entry for that page.

1.1.2 Using Select

The Mether kernel driver supports the *select(2)* system call. The interpretation is somewhat non-standard. A select which returns "write" status means that a page has just been unlocked by someone, so that the user-level server can take it if it is needed somewhere else. A select which returns "read" status means that someone wants a page. The determination of which page is wanted or has been freed is made by looking through the kernel data structures defining the pages. There are several pages of dynamically allocated kernel data structures which are accessible via the mmap system call. Because of the way the structures change state, it is safe for the user level process to examine and trust the state information found therein. Any change to the state is accomplished via ioctl system calls as described above.

We next discuss the user level server and the protocol it uses.

1.2 User Level Server

The user-level server runs as an event-driven loop. Events are IP-level User Datagram Protocol (UDP) messages and mether page-wanted/page-freed events detected via a *select(2)* system call, or a 100-millisecond timeout on the call. Every 100 ms. the server scans its internal lists of page state and examines the kernel driver Mether page descriptors.

While communications between the current set of user-level servers is via UDP there is no fundamental reason that UDP be the only protocol used.

There are currently three Mether packet types.

The first packet type is a request. If the user-level server sees that a page is wanted, it issues the packet shown in Figure 4. The packet's data consists of nothing more than an address in the Mether address space. There will be one or more packets containing the page returned in response to this packet. On the Sun, for example, eight packets must be returned. Since the UDP available on most BSD Unix systems will not accept an 8 Kbyte UDP message, the user-level servers must do fragmentation and re-assembly.

[4] The function used is hat_unload.

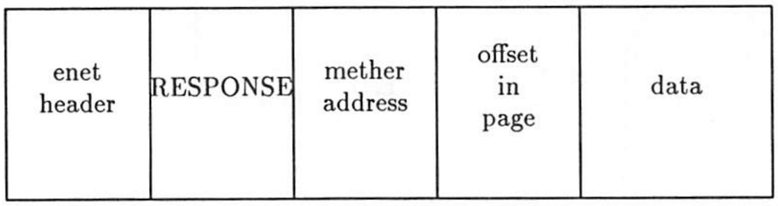


Figure 5: The response packet

There are two versions of the request packet. The first uses the return address of the requester. In this way only the requester need process the packet.

The second packet type, show in Figure 5, is a response. It is about 1 Kb long. It has a return address and a copy of a portion of the page. On a Sun system there are eight return packets for each request. We are currently evaluating the effectiveness of having responses always go to the broadcast address and having a 'watcher' keep track of the state of all the pages.

If the reliability of a TCP connection is desired then a different user-level server can be used, such as the one described in [5].

Note that none of the messages are acknowledged. There is a reason for the lack of acknowledgment which characterizes all Mether operations. On the original MemNet there is no acknowledgement either. There is a significant performance gain to be realized by taking advantage of the low error rate of Ethernet networks. Rather than using traditional error checking mechanisms as are found in TCP/IP, MemNet and Mether depend on the forward error correcting mechanisms supported by the hardware.

2 Design Issues

The design of Mether was driven by both the technical realities of implementing DSM in a system that was never designed for it (Unix), and the political realities of adding an experimental system to workstations being used by researchers for day-to-day computing activities such as mail, text editing, and program development. We needed to use a large number of machines, and such a large number is available to us only on production networks. We discuss both the technical and political considerations below.

2.1 Technical Issues

The first issue to be dealt with was the one which most impacted the design. The question of which operating system to use had been pre-determined: we knew that we would be using some variant of BSD Unix, either the SunOS, Ultrix[5], or Mt. Xinu[6] flavor. The question that drove all other design decisions was how much to modify the kernel. Were we willing to completely redo the paging functions, thus providing a completely transparent paging system? Could we get away with not changing the kernel at all, and allow the user program to trap faults and drive the paging via mmap(3)?

The trade-offs were explored, and in the end we decided not to modify the kernel fault handling code. The reasons were both technical and political. There are many and varied subtleties in the fault handling code in the kernel.

[5]Ultrix is a trademark of Digital Equipment Corporation

[6]Mt. Xinu is a trademark of Mt. Xinu Inc.

It was at that point that we discovered the SunOS 4.0 memory model. The SunOS 4.0 memory model is one of paged segments. Each segment has its own fault handlers. For mmap devices, the fault handler calls the device's mmap function when the user accesses a page for the first time. The implication was that fault handling was done for us. We got all the benefits of modifying, e.g. 4.3BSD, with none of the disadvantages. At this point we restricted our scope to SunOS 4.0 based machines, which was an acceptable decision as they far outnumber any other machine at our site.

The single hardest problem in using SunOS 4.0 is that very little information is available on how to use the new memory model. The new model in our opinion represents a fundamental improvement, comparable to the addition of paging to the Unix kernel. The only way we were able to determine how to use it was to buy sources and hunt through them. We hope to see this situation change.

2.2 Political Realities

It may seem strange to have a discussion on politics in a technical paper, but the fact is that in today's workstation environment politics have an unavoidable impact on a technical design. The reason is simple: in most people's minds, the network is *not* the computer. The computer is the workstation on their desk, and most people think of it as a stand-alone VAX[7] with a thin link to some nebulous collection of other computers. Never mind that their disk blocks and many other resources are provided by the network; if you are contemplating doing anything that impacts their machine they are likely to be quite unhappy. A resource such as Mether may improve the global environment for all users, but typically people only consider the machine on their desk.

As a result the Mether design conforms to several political necessities.

- The kernel driver must be very small and simple. There is no possible justification for increasing someone's kernel size by, say, 100 Kbytes.
- Any kernel structures must also be small and if possible dynamically allocated so that they do not consume space unless the driver is allocated. We use an array of page-descriptor structures each of which has a pointer to a page.
- The driver must never, ever take the kernel down. We have seen device drivers that panic on encountering the most basic inconsistencies. Panicing is unacceptable.
- Any user must be able to shut the driver down at any time. All resources allocated to the driver must be freed.

Having to allow politics to affect a technical design is distasteful. Unfortunately it is a necessary part of any system which wishes to use many workstations.

3 Summary

Mether is an implementation of DSM over Ethernet. The system runs on Sun workstations. The overhead for access to a page not on the local processor is about equivalent to a page fault. The system has recently come into general use, and performance results are very favorable.

Many of the interconnection networks being built for large numbers of processors exhibit the variable latency exhibited by Mether. Learning how to structure algorithms on such a network is an interesting research area and one which we expect to use Mether to explore.

[7]VAX is a trademark of Digital Equipment Corporation

3.1 Acknowledgements

Ron Minnich wishes to thank Maya Gokhale for reviewing early drafts of this paper; Leroy Fundingsland of SRC for the technical discussions of the shared memory implementation on Ultrix; and Gary Delp and Dave Farber for getting him started in this area.

References

[1] Gary S. Delp, Adarshpal S. Sethi, and David J. Farber. *An Analysis of Memnet – a memory mapped local area network.* Technical Report, University of Delaware, Department of Computer and Information Sciences, 1986.

[2] Gary S. Delp, Adarshpal S. Sethi, and David J. Farber. *An Analysis of Memnet: An Experiment in High-Speed Memory-Mapped Local Network Interfaces.* Udel-EE Technical Report 87-04-2, Department of Electrical Engineering, University of Delaware, Newark, Delaware 19716, April 1987.

[3] David Farber and Gary Delp. All systems in sync. *Unix Review*, 7(2):72–77, February 1989.

[4] A. Forin, J. Barrera, and R. Sanzi. The shared memory server. In *Usenix- Winter 89*, pages pp. 229–243, Usenix, February 1980.

[5] Don Libes. User-level shared variables. In *Usenix- Summer 85*, Usenix, 1985.

Distributing Processes in Loosely-Coupled UNIX® Multiprocessor Systems

F. Bonomi, P. J. Fleming and P. D. Steinberg

AT&T Bell Laboratories

ABSTRACT

We describe a method for distributing processes in a loosely-coupled UNIX® computing environment. Focusing on the `exec()` system call in the AT&T 3B4000 Computer System running the UNIX System V Release 3.1.5 Operating System, we discuss design and implementation techniques for process migration, the role of stub processes, as well as a novel load-balancing process assignment algorithm. We explain the stages of the `exec()` system call and show the method for migrating a process by constructing its new image on a different processor with the cooperation of all other processors comprising the extended process. Selection of the recipient processor is critical to load-balancing, so we discuss an algorithm that is an adaptive version of the Join-the-Biased-Queue policy. The algorithm utilizes a combination of the available instantaneous information about the number of processes on each processor and periodically collected average CPU run-queue length information as indices of load to make the assignment decision. A performance comparisons of three different process assignment algorithms is included to demonstrate the efficiency of our approach.

1. INTRODUCTION

This paper describes a method for distributing processes among processors, including load balancing considerations, in a loosely coupled computing environment. The work presented here was based on the AT&T 3B4000 Computer System running UNIX® System V Release 3.1.5. In this implementation process migration, which is the act of moving a process from one processor to another, only occurs during the `exec()` system call, and we do not discuss the more general case of dynamic migration. We begin by providing an overview of the aspects of the 3B4000 Computer System architecture that are relevant to process distribution and balancing workload. Next, we provide details of the `exec()` system call which is the focal point of process distribution. In subsequent sections we present the load balancing, process-to-processor assignment algorithm and the techniques used in its implementation. The paper concludes with performance comparisons of this algorithm with different classes of assignment algorithms on the 3B4000 Computer.

2. AT&T 3B4000 COMPUTER SYSTEM ARCHITECTURE

This section describes the aspects of the AT&T 3B4000 system that are relevant to the load balancing problem. An overview of the system hardware and software architectures is presented.

2.1 Hardware Architecture

The AT&T 3B4000 is, from a hardware point-of-view, a network of computers with closely cooperating kernels interconnected by a high speed bus. The individual processing elements are of several types connected together on a very high bandwidth system bus called the ABUS. The key observation about the hardware is that it is a highly coupled heterogeneous system containing adjuncts of varying speeds.[1] Each of the different processor types is specialized for certain functionality (e.g. disk control, high speed networking, etc.), but all run the UNIX System kernel and execute user processes. See reference [1] for a

1. The system architecture allows adjunct processors to have different instruction sets; however, all processors in the 3B4000 use the same instruction set.

detailed description of the hardware components of the system.

2.2 Operating System Architecture

The AT&T 3B4000 is a multiprocessor system where each adjunct processing element has the capabilities of a stand-alone computer, except for reliance on a *Master Processor* (MP) for global process management. This differentiates the AT&T 3B4000 from the general class of loosely-coupled networks of computers in which no single processor has global process information. For a more comprehensive treatment of the software architecture, see [2] and [4].

Each adjunct runs a separate copy of the UNIX Kernel in which accesses to main memory are local. The basic mechanism used in the operating system to implement the distributed operating system is the remote procedure call. Embedded within each system call is a filter function that determines whether the resource being requested is local to the *Processing Element* (PE) on which the system call occurs or remote. Here "resource" refers to file descriptors, message queues, pipes, etc. If the resource is local, the local kernel continues by way of the standard UNIX Operating System implementation. If it is remote, the request is packaged and sent, by way of kernel-to-kernel protocol, to the appropriate PE. The request will be sent directly to the agent of the requesting process on another PE, called a stub. A stub is a sleepable, dispatchable portion of an extended process that resides on an PE or the MP remote from the main user portion of the process.[2] The user process executes both user and system code that services local system calls while the stub process executes only the system code that services remote system calls. The kernel-to-kernel protocol provides a *virtual circuit* between the user portion of the process and each of its stubs. Thus a remote request, such as `lseek()`, will be transferred to the stub on the PE that owns the open file, where the standard `lseek()` kernel routine is called.

The `fork()`, `exec()` and `exit()` system calls are modified to account for the *extended* nature of a process, which includes the normal user portion, the stubs, and the virtual circuits that connect them. The `fork()` system call duplicates an extended process, and `exit()` tears one down. Since new process images are created during `exec()`, this system call plays a special role in process distribution and load balancing. We shall return to this subject in later sections.

There are two observations about the AT&T 3B4000 Operating System that are key to understanding process distribution and load-balancing. First, the fact that every user process **must** have a stub on the MP means that there is information local to the MP indicating the location of the user process image. Second, the MP receives a sanity message at regular intervals from each PE to make the MP aware of the ability of an PE to execute a process. Data related to the load of each individual PE can be included in these sanity messages with negligible additional overhead. Implicit in this discussion is the fact that there is no mechanism for migrating a process after `exec()` except to re-`exec()` it.

2.3 File System Distribution

The file system is identical in logical structure to the file system of the standard uniprocessor UNIX operating system. In particular, it is a single hierarchical file system; all PEs see the same file system structure regardless of the adjunct on which they reside, and the pathnames are unique across the entire system. Although the *logical* structure of the file system has been preserved, the system allows different parts of the file system to be maintained by different adjuncts; any process executing on any adjunct can locate any file with only a pathname.

The file systems on the PEs are mounted on directories in the root file system of the MP. When a mount point is traversed during pathname resolution, a remote indicator is checked, and the possible redirection of

2. A stub consists of a user block and the associated kernel stack. It occupies the same slot in the process table of the local processor as the user process does on its PE. A stub has a priority, vies for system resources like a user process, and is dispatched from a local kernel's run-queue.

the request takes place, including the establishment of new stubs and virtual circuits as required. The only restriction on mounting file systems is that the file system to be mounted must be mounted on a directory resident on the MP.

The primary implication of the AT&T 3B4000 file system structure on load-balancing is the fact that a process can place a processing load on a remote adjunct by requesting access to data or a peripheral mounted on that remote adjunct. For example, in the case of a file access, the processing load on the remote adjunct comes from three sources: message protocol processing, the stub process executing the appropriate system call, (e.g. `read()`), and, if the data is not in a buffer cache, the disk driver must run. This is load that the MP is unaware of since once a file is opened, the MP is not involved in subsequent accesses to that file. Other resources (e.g., message queues and remote *a.out* file demand paging activity) exhibit similar latent work characteristics.

Special device files associated with adjuncts are accessed through an adjunct-specific in-core file system. Most significant among these files are disk/tape special device files and tty/networking special device files. Communications processors tend to have several static `getty` processes that are associated with unutilized tty lines. These processes, which sleep and do no work, are taken as active processes by the MP and could be assumed to be contributing to the system's overall load. The important observation here is that processes do not generate equal amounts of work and this causes imbalances when using a process as the unit of work.

3. *THE IMPLEMENTATION OF exec()/exece()*

In the AT&T 3B4000 System, the `exec()` system call is used to distribute processes among the processors. Basically, the old (exec'ing) process is torn down and the new (exec'd) process in constructed in its place, potentially on a different processor. When a process performs an `exec()` there is an absolute minimum of information that has to be moved since *text*, *data*, *bss*, stack, and state information can be reconstructed for the new process image rather than moved.[3] Consequently, this provides the most efficient point at which to migrate the process in a loosely coupled system. Note that the `exec()` system call provides the sole mechanism for distributing processes in the AT&T 3B4000, and there is no general purpose, dynamic migration primitive.

3.1 The Modified Design of exec() for the AT&T 3B4000

There are five logical Processing Element (PE) "types" which may be involved in processing some part of an `exec()` system call in the distributed model. On any given `exec()` call, any of the five types may actually be subsumed by one or more of the others. Any other configuration can then be viewed as a simplified special case of this more general configuration. The five types follow. (1) The *Olduser PE* is the owner PE of the process that issues the `exec()` system call. We call this process the *olduser* process. (2) The *A.out PE* is the owner PE(s) of the *a.out* file(s) specified in the `exec()` system call. The *a.out* PE can be as many different processors as are required to locate both the primary and secondary[4] *a.outs*. (3) The *Newuser PE* owns the process which results from the completed `exec()` system call. We call this process the *newuser* process. This may be the same as the olduser PE, and if so, this is then called a *local exec*. (4) The *Process Manager* (equivalent to the MP) is the PE which selects the newuser PE. Every process is required to have a stub on the MP. (5) *Stub PEs* are those PEs with stubs of the olduser process (other than the olduser and newuser PEs).[5] The *a.out* PE(s) and the Process Manager PE are always considered stub PEs, provided these are not the olduser or newuser PEs.

3. With the exception of *text*, processes do not have remotely mapped pages in their virtual address space.

4. Here, the word *primary* refers to the actual *a.out* file that is the object of the `exec()` system call. The term *secondary* is used in reference to any *a.out* file specified for inclusion in the resultant process address space. A secondary *a.out* file is, in fact, a shared library. More than one secondary *a.out* file can be required for inclusion by a primary *a.out* file.

5. Recall that a stub will be required on each PE that is servicing a remote resource of the process (e.g. an open file).

The design of the `exec()` system call for the AT&T 3B4000 divides exec processing into three stages.

I. *Pre-Exec Stage*: During this stage the newuser PE is selected and the olduser's address space is torn down.

II. *Migrate Stage*: This stage facilitates movement of the olduser process from the olduser PE to the newuser PE. (In the case of a *local exec*, this stage is not visited.)

III. *Exec Proper Stage*: The newuser process's address space is constructed during this stage.

In the vernacular of the stages just given, the high-level design for `exec()` on AT&T 3B4000 can best be described as follows.

```
Pre-Exec Stage (on olduser PE)
IF olduser PE ≠ newuser PE THEN Migrate Stage
Exec Proper Stage (on newuser PE)
```

An important observation is that the *Exec Proper Stage* must work the same way, regardless of whether or not the *Migrate Stage* is performed.

Figure 1 shows the relationships between the objects and stages of exec. The boxes represent PEs, and the lines represent communication among them. The Roman numerals and letters are keyed to the descriptions of the stages given below.

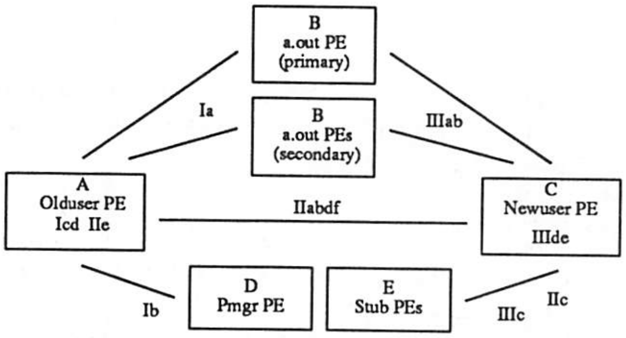


Figure 1. Structure of *exec()* in the AT&T 3B4000.

3.1.1 Pre-Exec Stage (Stage I)

[Ia] *Get A.out Header Info.* At the beginning of an `exec()` call, the olduser process must be furnished with a stub on the primary *a.out* PE, the *a.out* file must be located, and its header information loaded into the `u_exdata` area of the olduser process' u-block.

Information in the primary *a.out* header is used during the selection of the target processor (newuser PE) for the exec, and is later installed on the newuser PE. This information contains: (1) the size of the *a.out* file (*text* + *data* + *bss*); (2) the type of cpu (as indicated by the magic number of the *a.out* file and its flag word); (3) any user-provided PE assignment information explicitly indicated in the *a.out* file[6]; and (4) the number of shared libraries and, if not zero, the shared library section which contains the pathname(s) of secondary *a.out* file(s).

6. The AT&T 3B4000 provides a specific system call that can be used to provide process-to-processor assignment direction to the operating system. This information can be recorded in a special section of the *a.out* header, or it can be specified dynamically via information stored in the olduser process's u-block.

Since multiple *a.out* files must be managed during exec and the subsequent memory management issues, an array of pointers is dynamically created to account for the files.[7] This array contains an inode pointer if the *a.out* file is local or a virtual circuit pointer to the remote PE servicing the inode. Stubs follow a similar convention except that their arrays contain only local inode pointers.

Stubs are created as necessary on PEs controlling secondary *a.out* files; the *a.out* files are located, and their headers are read into a dynamically allocated memory slate.[8] Like the primary header, information from each secondary header must be transmitted back to the olduser PE: This consists of the size of the secondary *a.out* file (*text* + *data* + *bss*). This information is accumulated along with the sizes of the primary *a.out*.

[Ib] *Select Newuser PE.* The olduser PE communicates the above size information and any specific PE assignment criteria (optional) to the MP. The MP selects the identity of the newuser PE by applying the load balancing algorithm, discussed later, within the user-specified assignment constraints (or validates an explicit assignment, if one is given). The MP returns the identity of the newuser PE to the olduser PE.

[Ic] *Get Exec() Args.* The olduser PE next copies the arguments from the `exec()` call (and any environment variables) from the olduser's address space into a dynamically allocated memory slate and formats them into the newuser's initial user stack.

[Id] *Tear Down Olduser's Address Space.* The olduser PE now destroys the olduser process's address space, releasing its resources to the system. After completing this step, we have passed the *exec commit point.*[9]

3.1.2 Migrate Stage (Stage II)

[IIa] *Create Stub on Newuser if Necessary.* If the newuser PE is different from the olduser PE, the operating system must guarantee that the olduser process has a stub on the newuser PE, to allow the communication necessary for migration.

[IIb] *Emigrate.* The critical part of process migration is the transmission of the migration information from the olduser PE to the newuser PE. This provides the information necessary to transform the stub on the newuser PE into a viable user-portion of an extended process. Here, viability is defined to be exitability, which means that the newuser process has all the information necessary to at least be able to exit gracefully on the newuser PE. Once this viability is achieved, the migration is said to have passed its *exit commit point*, and, as far as resource cleanup is concerned, the former stub on the newuser PE really is the user-portion of the extended process.

The migration information includes: (1) reconnection data for the stubs and open files of the process, so that these can be reattached to the new user-portion of the process on the newuser PE; (2) the locality of the *a.out* files by PE location; (3) crucial data from the process's proc-table entry, such as its priority, nice value, process group id, parent process id, and flag word; (4) crucial data from the process's u-block, such as its user id, group id, current directory, private root directory, new argument pointer, and various time-keeping fields; and (5) the olduser and newuser *a.out* file lists.

7. Actually, two arrays are maintained. One array keeps track of the olduser's existing *a.out* file information and the second array maintains information about the newuser process being constructed.

8. Since any PE may control any combination of primary and secondary *a.out* files, the locality into which the header information for a secondary *a.out* file is read can not be the `u_exdata` area of the u-block.

9. The "exec commit point" is the place in the exec procedure after which it is not possible for a failure to return an error to the olduser. Failure after exec commit results in a termination, via *exit()*, of the process.

[IIc] *Immigrate.* The stub on the newuser PE receives the migration information from the olduser, and acts appropriately, installing the data contained in the message into the newuser's proc-table entry and u-block, and issuing "reconnect" requests to the process's pure stubs,[10] reattaching them to the newuser PE. After this is completed, the newuser process is considered viable. At this point, all portions of the extended process consider the user portion of the process to be located on the Newuser PE so that items such as signals can be routed correctly (signals must always be sent to the user-portion of the extended process).

[IId] *Transfer Exec() Args.* After the olduser PE has sent its migration information, it sends the process's `exec()` arguments (obtained from the olduser process in Stage I). After the newuser PE has built up the newuser process to the point of being exitable, it installs the arguments into the newuser's address space as its initial stack. Fortunately, the starting address of the user stack is the same on all processor types so no translation of argument or environment pointers is necessary.

[IIe] *Turn Olduser into Stub.* After the newuser PE has successfully transformed its stub into the new user-portion of the extended process, it informs the olduser PE, which responds by transforming its old user-portion of the extended process into a pure stub. This involves adjusting certain fields in its proc-table entry and u-block to standard stub values, and calling the common stub entry function. The stub spends the rest of its existence here unless some subsequent `exec()` system call brings the user portion of the extended process back onto this PE.

[IIf] *Get Signals.* The last step involved in a migration is to copy and clear the signal word of the olduser process. During the course of the migration, signals destined for the process may arrive at either the olduser PE or the newuser PE, and be posted. Thus, the olduser PE may have signals which need to be forwarded to the newuser PE and therefore the olduser PE must now send its signal word to the newuser PE, where the two values are bitwise OR'd together and stored for the newuser process.

At this point, a minimal, viable process (proc-table entry and u-block, with re-attached stubs and files) has been built on the newuser PE.

3.1.3 Exec Proper Stage (Stage III)

[IIIa] *Transfer A.out Header Info.* The first step of this stage is to transfer the *a.out* header information, (gathered in Stage I) from the *a.out* PE(s) to the newuser PE, together with other information (mode, user id, group id) needed during Stage III processing. Only the *a.out* header information for the primary file is required at this time. Secondary header information is acquired as needed in the next step.

[IIIb] *Get Executable Files.* With the information obtained in step IIIa, the newuser PE now builds the newuser process's address space (*text*, *data*, and *bss*) by mapping the various sections from the primary and secondary *a.out* files into regions of the newuser's address space. Pages are acquired by the executing process through a demand load strategy.

[IIIc] *Close-on-Exec Processing.* The open files (local and remote) which are marked close-on-exec can now be closed, and the olduser *a.out* files can be released (`iput()`).[11]

[IIId] *Final Touch-up.* Any signals set to be caught by the olduser process must be reset to the correct (default or ignore) action for the newuser process. Hardware specific initialization can also be performed before returning to the new user process the first time.

10. Stubs which are otherwise uninvolved.

11. In a system configured with a large number of PEs, and where the maximum number of files a process is allowed to have open is large, close-on-exec processing could potentially be time-consuming. In practice, however, the close-on-exec attribute for files is the exception rather than the rule.

[IIIe] *Return to User*. At this point control can finally be passed (or returned) to the entry point of the newuser process. The routine within the kernel responsible for passing (or returning) control to a user process is `systrap()` which simply checks for pending signals and "returns" to user level.

Given this overview of exec processing, we now turn to the issue of load balancing considerations in selecting the newuser PE.

4. PROCESS-TO-PROCESSOR ASSIGNMENT ALGORITHM DESCRIPTION

The most important performance indexes for a load balancing algorithm on a multiprocessor computer are the user-perceived performance measures usually described in terms of response time and throughput. Unfortunately, these indexes are too high-level to be of immediate use to an automatic load balancing algorithm.

The underlying philosophy of our approach is that there is some internal state of the system that needs to be maintained in order to provide optimal (or near optimal) user-perceived performance. Maintaining the system in such desirable internal states involves maintaining balance in some easily quantifiable load measure such as queue lengths or processor idle times.

The Adaptive Join the Biased Queue (AJBQ) algorithm described below attempts to balance CPU run-queue length normalized by processor speed. A more thorough treatment of the mathematical aspects of the algorithm is contained in [3].

The following table contains the notation used in this section.

M	number of processors in the system labeled $i=1$ through $i=M$.
μ_i	processing speed of the i^{th} processor.
$N_i(t)$	number of processes assigned to processor i at time t.
t_k	clock time at the beginning of the k^{th} measurement interval.
τ	length of the measurement intervals. (i.e. $\tau = t_{k+1}-t_k$). Refer to Figure 2.
ν	measurement sampling rate. ν^{-1} is the time between samples. Refer to Figure 2.
$n_{i,k}$	average run-queue length at processor i during the interval (t_k, t_{k+1}).
$\hat{n}_{i,k}$	estimate of $n_{i,k}$ based on samples of the run-queue length of processor i during the k^{th} measurement interval taken ν times per second.
$c_i(t)$	primary index of load of processor i at time t, referred to as the *biased-queue-length*.
$\Delta_{i,k}$	queue-length bias term for processor i during the k^{th} measurement interval.
$\overline{\Delta}$	upper bound on the value of the $\Delta_{i,k}$, $-\overline{\Delta}$ is the lower bound.
S	set of feasible bias terms, i.e. the set of all $(\Delta_1, \ldots, \Delta_M)$ such that $\sum_i \Delta_i = 0$ and $-\overline{\Delta} \le \Delta_i \le \overline{\Delta}$.
π_S	orthogonal projection operator into S, i.e. $\pi_S(\Delta_1, \ldots, \Delta_M)$ = the point in S closest to $(\Delta_1, \ldots, \Delta_M)$.
α	scalar gain factor.
$\varepsilon_{i,k}$	deviation of the estimated average normalized run-queue length of processor i over the k^{th} interval from the average over all the processors of the same quantity.

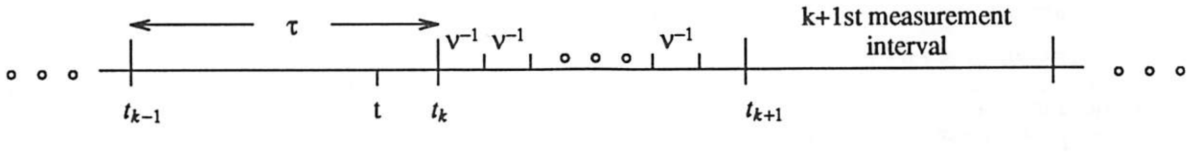


Figure 2. The Measurement Interval

There are three equations that describe the AJBQ algorithm from a mathematical point of view. The primary index of how much work the i^{th} processor has at time t is called $c_i(t)$ and is computed as follows

$$c_i(t) = N_i(t) + \Delta_{i,k}\ ,\quad t_k \le t \le t_{k+1}\ ,\ i = 1, 2, ..., M\ . \tag{1}$$

Notice that $c_i(t)$ is a combination of instantaneously accurate measure of load ($N_i(t)$), and load history ($\Delta_{i,k}$) on each processor. The instantaneously accurate data provides stability to the algorithm and damps the effect of negative feedback from the history data. This is an adaptive version of the Join-the-Biased-Queue policy first studied by Yum [5], who considered only static bias terms. The $\Delta_{i,k}$ terms evolve from interval to interval as follows

$$\Delta_{i,k+1} = \pi_S\left[\Delta_{i,k} + \alpha\varepsilon_{i,k}\right]. \tag{2}$$

The bias term $\Delta_{i,k+1}$ is an adjustment to $\Delta_{i,k}$ based on $\varepsilon_{i,k}$, a function of the state of the run-queues on each of the processors over the k^{th} interval. Since this adjustment may cause the new Δs to be outside the feasible set S (see above notation), the projection operator, π_S is applied. The projection is needed to control drift. This will be explained more thoroughly in the implementation section.

The next equation presents the calculation of the $\varepsilon_{i,k}$.

$$\varepsilon_{i,k} = \frac{\hat{n}_{i,k}}{\mu_i} - M^{-1}\sum_j \frac{\hat{n}_{j,k}}{\mu_j}\ . \tag{3}$$

Note that the first term in Eq. (3) is the estimate of the average run-queue length on processor i during the k^{th} interval normalized (or, divided) by the speed of processor i. Subtracted from this is the average of the normalized estimated average run-queue lengths over all the processors in the system. So the amount the bias terms (Δ) change from interval to interval depends on how much the normalized average run-queue-lengths differs from the average over all the processors.

The bias terms will stabilize when

$$\frac{\hat{n}_{i,k}}{\mu_i} = \frac{\hat{n}_{j,k}}{\mu_j}$$

from interval to interval. In this case, we say the bias terms have converged. Under normal operating conditions the bias terms should converge. If the load somehow suddenly shifts (e.g. from terminal I/O intensive to computation intensive) it will generally take two to four measurement intervals (one to two minutes) before the bias terms settle down to their new values. The values of τ, ν and α play an important role in the convergence of the bias terms. Both simulation and measurement have shown that τ=30 sec., ν=10 msec. and α=1 are reasonable choices and the performance of the algorithm is robust around these values.

5. IMPLEMENTATION OF PROCESS-TO-PROCESSOR ASSIGNMENT

This discussion assumes some familiarity with the fundamentals of UNIX Operating System internals in standard uni-processor implementations. There are three basic phases involved in implementing the algorithm: (1) gathering statistics about each of the M PE's run-queue lengths, (2) periodically computing the bias terms Δ, and (3) applying the bias values for asynchronous process assignment considerations.

5.1 Collecting Statistics

We need to accumulate the average run-queue lengths of the various PEs over successive time quanta (of length τ). The measurement periods are reasonably long (30 seconds) to assure the accuracy of the average load assessment. For the average to be accurate, even over a significant period of time, all of the PEs in the complex must report to the MP at regular intervals. The MP is responsible for making the process-to-processor assignment decisions, and as such, it is the focal point for the gathered statistics. Regular information collection is possible through the implementation of a periodic sanity reporting mechanism that the MP uses to monitor the PEs. This sanity reporting mechanism uses the "I am alive" technique, and is implemented in such a way as to guarantee that each adjunct in the system sends a sanity message to the MP every 2½ seconds. Therefore, the MP will receive updates from a PE about 12 times over the course of a 30 second time quantum.

The process dispatchers of all PE's kernels keep a running count of the number of processes on their run-queue. For each PE i, on every clock tick (the clock tick frequency was referred to as ν above, and is every 10 milliseconds) the PE adds the length of the run-queue, $q_{i,k}^{j}$ to an accumulating bucket and increments a counter to record the number of samples taken. Every 2½ seconds, a sanity message is forwarded to the MP. This message contains the average run-queue length over the previous 2½ second interval, which is obtained by dividing the run-queue accumulation by the number of samples. This gives us a 2½ second component of $\hat{n}_{i,k}$ (denoted here by $\hat{n}'_{i,k}$), since it is only computed for a portion of τ (recall from Figure 2 that τ is the entire 30 second interval). The accumulator and counter are cleared and a new 2½ second cycle begins.

Upon receiving the sanity message, the MP extracts the average from the message, adds it into a PE-specific accumulation bucket, and increments a counter of the number of sample averages accumulated. The average and counter are stored in the global PE configuration table. The MP keeps similar statistics about its own run-queue length; however, obviously no sanity message is required. Thus, for the computation of a bias value over the full 30 second interval, samples of the run-queue length are accumulated and instantaneously available to the MP. When the quantum has elapsed, an average of the $\hat{n}'_{i,k}$ components can be computed to obtain a working value for $\hat{n}_{i,k}$ over the entire interval τ.[12] This is the estimate for the asymptotic average CPU run-queue length for PE i over interval k. Here, the $\nu\tau$ term is approximated over the two-staged averaging approach through the accumulation of sample counts. After bias computation, the MP clears the counter and accumulated average to begin a new cycle.

5.2 Computing the Bias Values

Every τ seconds the algorithm used to compute the bias factors, $\Delta_{i,k}$, is invoked on the MP via callout processing. As was discussed above, it computes an average, $\hat{n}_{i,k}$ of the component average run-queue lengths for each PE, $\hat{n}'_{i,k}$, and clears the average accumulator and sample counter for the next quantum.

A two pass approach with an optional third pass is used for the computation of $\Delta_{i,k}$ for PE i over interval k. The first pass computes the estimate for the asymptotic average CPU run-queue length, $\hat{n}_{i,k}$, as described above and adjusts this value by the service rate, μ_i for each PE, i (effectively, the minuend in Eq. (3) is computed). The first pass also accumulates the sums of these terms for use in a later computation of the subtrahend in Eq. (3) (this summation is simply divided by M).

The second pass computes the $\varepsilon_{i,k}$ (see Eq. (3)) for each PE i and uses these values to compute the bias, $\Delta_{i,k}$, for each PE to be used over the $k+1$ interval. The projection operator, π_S, is also applied to the bias value if a computed bias value falls out of the bias bounds ($-\bar{\Delta} \le \Delta_{i,k} \le \bar{\Delta}$, $\bar{\Delta} = \pm 100$). The intent behind bounding the bias terms is to keep the bias values from becoming very large and diminishing the adaptability of the algorithm for the other PEs. Without the bounds, the bias values could continue to grow (in absolute value) for load imbalances that process assignment is unable to correct (e.g., stub activity on the MP). In other words, the second pass computes Eq.(2) for each PE.[13]

12. One problem that had to be overcome in implementing the algorithm stems from the fact that floating point arithmetic is not supported in the UNIX Operating System Kernel. To compensate, the operands of the computations are scaled to maintain precision and then integer arithmetic is used. The level of precision is currently two decimal places (i.e., all dividends are multiplied by 100). This results in truncation (not round off) and has some other unfortunate side effects that had to be addressed during computation of the bias values.

13. Note that the bias is positive for PEs that have run-queue lengths greater than the average of all PEs and it is negative for PEs that have run-queue lengths less than the average of all PEs.

The third pass is only needed when all computed bias values do not sum to zero. This situation can occur for two reasons. The first reason is when one or more PEs is overworked, relative to the other PEs, such that it exceeds the upper bias bound (*i.e.*, $\Delta_{i,k} > \overline{\Delta}$), and the π_S, had to be applied in the second pass to force it to remain in bounds. The second source of error to the bias terms is truncation error resulting from the integer division and limited precision forced upon the algorithm. Adjusting the bias values so that they sum to zero prevents the bias terms from drifting due to this computational error.

The bias terms are forced to sum to zero by applying a correction term to the computed bias of each PE still within the bounds such that convergence and stability are ensured.[14] During the second pass, the $\Delta_{i,k}$ values were summed for each PE i, and a counter was incremented for each PE that is still within the bounds ($< M$). In the third pass, the deviation of the bias terms from zero is divided by the number of PEs within the bounds giving us a correction term. This correction term, which is the deviation of the bias summation averaged over the PEs that can tolerate adjustment (i.e., those PEs that are within the bounds), is then added to the bias of each of the PEs still within the bounds. This causes all of the bias values to sum to zero.

Note that it is possible for the application of the correction factor to cause a PE's bias to exceed the upper or lower bounds. One possible solution is to repeat the entire bias computation for those PEs still within the bounds; however, in the interest of efficiency, this step was omitted from the implementation. Instead, the projection operation is simply applied. The additional overhead is not necessary since we have eliminated the possibility of the bias terms drifting due to computation errors.

5.3 Applying the Bias to Processor Assignment

The bias values, $\Delta_{i,k}$ are used by the load balancing algorithm over the quantum k to weight the individual process counts, N_i, for each PE under consideration. The MP maintains N_i by incrementing the value for every `fork()` and `exec()` that causes a new process to be created on PE i. The value is decremented for each `exit()` or `exec()` that removes a process from PE i. Eq. (1) is applied each time an `exec(2)` system call is issued (*Stage Ib*) and, of the eligible PEs, the one with the smallest c_i value (least load) is selected for assignment.[15] An eligible PE is one that falls within any user specified criteria and has sufficient real and virtual memory to support the process.

The algorithm has several constants that can be adjusted to influence its behavior, and these values are maintained as `#define's` in a header file. The implementation of the algorithm presented here involves about 300 new lines of code, touching only routines which have a simple interface with the rest of the system.

6. PERFORMANCE COMPARISON

Extensive performance analysis results are presented in [3]. These results demonstrate that the AJBQ algorithm is (1) simple to implement with low processing overhead, (2) robust with respect to workload, file system distribution and the configuration of processors of different speeds and functions, and (3) provides improved user-perceived response times and throughput with minimal or no tuning required.

We compared two other seemingly reasonable load-balancing algorithms with the ABJQ algorithm. The Round-Robin (RR) algorithm simply assigns processes to processors in a cyclic fashion, first assigning to processor 1 then to processor 2 and so on. The RR algorithm was modified to exclude the MP, however, since it is bottleneck under heavy load. The Join-the-Shortest-Active-Queue (JSAQ) is the AJBQ with α, the gain factor, set to zero, so that the only assignment criterion is the number of processes currently

14. Only PEs within the bias bounds can tolerate adjustments.

15. Since the bias values are scaled upward by a factor of 100 to account for the absence of floating point, the process counts, N, must also be similarly scaled so that the two terms may be combined.

assigned to each processor ($N_i(t)$). Again, we have adjusted JSAQ to exclude the MP. In addition, a significant amount of pre-measurement experimentation was done to balance the file system accesses on the various PEs. The AJBQ algorithm looks even better compared to RR and JSAQ when there are imbalances in the accesses. This is what we mean by ***robustness*** of the algorithm.

A remote terminal emulation (RTE) tool was used to measure the response time performance of the three different load-balancing algorithms on an AT&T 3B4000 Computer System consisting of the MP and 8 PEs with various numbers of emulated users executing a mix of simple and complex commands. Table 1 contains a summary of those measurements with the results in seconds.

TABLE 1. 90th Percentile Response Time Comparison
(Each entry is the average and standard deviation (in parenthesis) of six measurements.)

Algorithm	Simple Commands			Complex Commands		
	120 users	160 users	190 users	120 users	160 users	190 users
RR	0.66 (0.36)	1.32 (0.36)	T-O[14]	12.4 (2.8)	26.7 (2.8)	T-O
JSAQ	0.83 (0.69)	1.64 (0.69)	2.20 (0.35)	20.1 (5.8)	48.2 (5.8)	60.1 (6.9)
AJBQ	0.70 (0.03)	0.91 (0.03)	1.28 (0.04)	12.4 (1.3)	19.7 (1.3)	42.1 (1.9)

The ABJQ algorithm is the clear winner at 160 and 190 emulated users, providing in excess of 40% improvement in the 90th percentile response time for simple UNIX commands such as `ls` and `date` and 30% improvement for complex commands such as `make`, `nroff` and `pr * > junk` over both RR and JSAQ. All three algorithms performed about the same with 120 users since the load was too light to differentiate them. JSAQ performed uniformly poorly because of the presence of a large number of `getty` and `sh` processes on the ACPs, effectively excluding them from being considered for assignment. It is possible that JSAQ is better than RR at 190 users, unfortunately, the RTE tests of the RR algorithm failed at 190 users without yielding any useful data, due to some very long response times which caused time-out. Note also the order of magnitude improvement in the standard deviation of the measurements for AJBQ. This reduction in response time variance provides a measure of fairness and predictability for the users of the system.

Figure 3 shows the results of a number of throughput tests based on a configuration consisting of the MP and 4 PEs. The AJBQ did slightly better than JSAQ and about 20% better than RR at peak throughput. As expected, JSAQ performs better in the throughput test than in the response time test since there are fewer `getty` and sh processes.

14. These tests did not complete successfully due to a large number of commands timing out (T-O) with response times greater than 900 seconds.

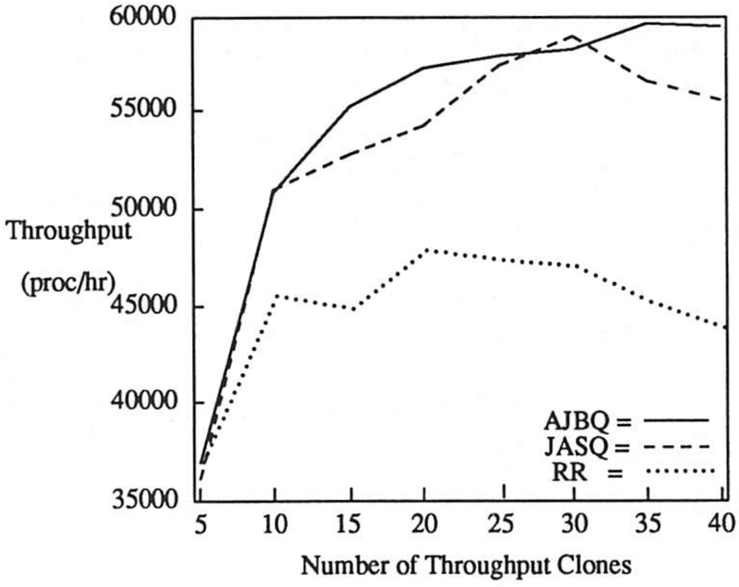


Figure 3. Throughput Comparison for AJBQ, JASQ and RR algorithms

7. ACKNOWLEDGEMENTS

Tom Bishop, Bob Fish and Walt Tuvell are primary designers of the AT&T 3B4000 `exec()` system call described in this document. We extend our gratitude to Jim Peterson, Brian Rodgers, George Shutack and Y. T. Wang for technical help and problem definition in development of the AJBQ algorithm. We also thank Brett Behm, who spent many hours in the lab doing performance measurements of the AJBQ algorithm prototype. Finally, we would like to thank each and every person in the AT&T 3B4000 project for their support and encouragement.

REFERENCES

1. AT&T 3B4000 Computer Software Architecture, Doc. No. 303-310, Issue 1, 1988.
2. AT&T 3B4000 Computer Hardware Description, Doc. No. 303-303, 1988.
3. Bonomi, F., Fleming P. J., and Steinberg, P. D., "An Adaptive Load Balancing Algorithm for a Class of UNIX Multiprocessor Systems," AT&T Bell Laboratories Technical Memorandum 45312-881211-01TM, December 11, 1988 (submitted for publication).
4. Opferman, D. C., "The AT&T 3B4000 Computer System," *AT&T Technology*, Vol. 3-3, 1988.
5. Yum, T. P., "The Join-Biased-Queue Rule and its Applications to Routing in Computer Communication Networks," *IEEE Trans. on Communications*, vol. Com-29, no. 4, pp. 505-511, April 1981.

Administration of Department Machines by a Central Group

Denise Ondishko
Unix Operating Systems Group Leader
University of Rochester Computing Center
727 Elmwood Avenue
Rochester, New York 14620
(716) 275-0345
dmo@cc.rochester.edu
ucbvax!ames!rochester!ur-cc!dmo

Topic Area: System Management

Abstract

Like most other universities, the University of Rochester has seen a rapid growth in departmentally owned machines running a version of the Unix® [1] operating system. Each department typically has a special task required of its machine. In addition, each department desires control over its own resources. By providing a centrally trained group of system programmers a university can avoid duplication of effort and education on the part of many departmental system programmers. However, the needs of the central programmer and the departmental programmer conflict: it would be easiest for the departmental programmer to make any changes at any time and it would be easiest for the central programmer if all of the machines looked identical and did not change unpredictably.

The University of Rochester Computing Center has developed a method of resolving these differences through a **contract of services and responsibilities** which specify areas of the operating system which must only be modified by the central system programmers and areas of the system which are the exclusive responsibility of the department programmer. The department programmer must conform the system usage somewhat to a central norm in order to get the benefits of central administration. However, unlike most centrally administered sites, system changes are initiated by the department administrator rather than by the central group. This paper will describe the keys to the success of the support contracts.

DEPARTMENTALLY EMPLOYED SYSTEM PROGRAMMERS

Departments which purchase their own machines typically have a special task required of the machines. The department can choose to purchase the best machine to perform the specialized task it deems most valuable and it won't have to compete with other users whose applications are not important to the department. For example, at the University of Rochester the Laboratory for Laser Energetics performs real-time digitization and image analysis; the Eastman School of Music uses its machines for digital-audio sound synthesis and production; and the department of Psychology performs lengthy runs of statistical analysis programs. The special purpose software packages on these machines are not sharable by many

[1] Unix® is a registered trademark of AT&T.

departments. In addition, each department desires control over its own resources. The immediacy of response from a departmentally employed system administrator is also very attractive to department users.

Departmental programmers vary widely in experience and knowledge of Unix operating system utilities. Some have experience maintaining other operating systems and are expected to pick up administration of a Unix operating system on the job. Others might be competent to maintain their system, but must spend practically all of their time developing and supporting local utilities. Most department programmers require some training in system administration and operating system programming. The novice group of department system programmers may require more training than most departments can afford so the programmer is forced to learn on the fly—often just maintaining the parts of the machine directly involved with the application the programmer is most familiar. The skilled programmer may perform system maintenance only when emergencies dictate—forgetting or deterring routine checks and preventive maintenance. Eventually, when department programmers must make changes beyond their capacity or time allotment, they will contract with a central programmer to perform the task. When a central programmer goes to work on these machines, the myriad of inconsistencies and left-over, half-done changes make the requested changes all the harder to implement.

CENTRAL MAINTENANCE

The departments of a university can avoid duplication of effort and of training on the part of many system programmers by sharing and training a central group of system administrators. In the past, when machine costs were prohibitive, centrally supported machines were shared by many departments. At this time, machines running a version of the Unix operating system are popular because of their low price to performance ratio. Meanwhile the cost of a salaried system programmer has gone up—threatening to match the old costs of a centrally shared machine.

Since the number of machines being maintained continues to increase rapidly, certain similarities, or consistencies, between all systems should exist in order to keep the number of a central programmer's work hours from increasing, also. Central system administrators should be able to automate many aspects of system administration, such as distributing software easily to file systems on all machines using *rdist(1)*, or other similar programs[2] and checking on irregular system changes. In addition, the central programmers should work with each of the departments in a consistent and efficient manner, having similar responsibilities towards each department.

Large installations which are maintained by a central group are usually comprised of a single model (the master) which gets copied around to all the remote hosts. In this way, a central group is maintaining only one machine, many times. This is the easiest way to maintain a large number of machines. The central group can then devote their time to security checks and tool-building, with part of the group performing hardware maintenance and trouble-shooting.

Departments are usually not in favor of this method of maintenance, especially if one of the reasons they got their computer was to gain control over it within the department. Also, only small groups of departments may agree on how its resources should be used, or may be willing to be led by a central group. Eventually the computing resources at the university are dispersed and central maintenance fails.

In simple terms, the needs of the central programmer and the department programmer conflict: it would be easiest for the department programmer to make any changes at any time and it would be easiest for the central programmer if all of the machines looked alike and never changed unpredictably.

[2] See Nachbar, Daniel "When Network File Systems Aren't Enough," USENIX 1986 Summer Conference.

OUR APPROACH

To accommodate the disparate needs of the department programmers and the group of central system administrators, a standard **contract of services and responsibilities** has been developed which a central group of system programmers can provide to the diverse groups and departments of the university. The key to the success of the support contract is that it provides for a consistent area for central maintenance and a separate area for local changes and improvements to be made. Clearly defined responsibilities between the department programmers and the central system programmer help rule out unauthorized and rogue changes.

THE UNIVERSITY COMPUTING CENTER

A few years ago the University of Rochester Computing Center (UCC) Unix operating systems programmers maintained a handful of general purpose timesharing machines which were used by many departments. Like most other universities, the University of Rochester has seen a rapid growth in departmentally owned machines, from diskless Sun Workstations®[3] to Alliant®[4] mini super computers.

The Unix operating systems group of the University Computing Center is a six-member group consisting of three Senior Operating System Programmers, a group leader (also a System Programmer), and two part-time student programmers (Technical Assistants). In addition, we have a consultant and six operators to help monitor the machines. Four machines are owned by the UCC and still fall under the description of centralized computing. Departments will pay the computing center for partial use of the machine, especially for classroom usage. These centrally shared machines are a Sun 3/280S with over 200 student and faculty users, an Alliant FX-8 with a research community of over 100 users, and two Vax® 11/750s with over 100 researchers and students.

The remaining 40+ machines are departmentally owned machines that the UCC Unix operating systems group has been asked to maintain. Three central programmers currently maintain over 43 machines, serving 15 user groups and departments. Some of the departmentally owned computers are located and maintained in UCC operations areas. Other machines are located in user areas within the various buildings on campus.

The University of Rochester campus LAN is a Class B network. The LAN is a fiber optic ethernet spanning many buildings and departments, and is largely IP based, but supports other protocols. The backbone network physically and logically connects departmental networks to each other and to wide-area networks. The Network group of the UCC (consisting of three engineers/programmers and a team of technicians) manages, designs and maintains the campus backbone network and network applications. The Unix operating systems group and the network group of the UCC frequently work together on projects and troubleshooting sessions.

COMMUNICATION

Each site, or machine, will have a primary system administrator from the central group and a co-administrator from the department. On the centrally shared machines two central programmers will serve as primary and secondary administrators. These two people work closely together to maintain a known status of all the work in-progress, current problems, and timely handling of user demands at each site by reporting all changes and plans to each other.

We communicate with our sites continuously in two modes: by polling and by receiving interrupts. Like the minute and hour arms of a clock, we poll our sites at two rates: every day we check on email from users and from automated programs, and every week we visit our sites for regularly scheduled maintenance

[3] Sun Workstations® is a registered trademark of Sun Microsystems, Incorporated.

[4] Alliant® is a trademark of Alliant Computer Systems Corporation.

sessions. Interrupts normally take the form of emergency calls—sometimes via phone calls and sometimes via a pager. The programmers in the central group rotate a pager on a daily basis. Since we are not always available at our office phones, this allows emergency calls to reach someone quickly.

Constant Problem Consulting—responding to user problem reports and how-to's.

Users will typically phone or email system staff or the consultant if they have questions or problems with a system. We encourage users on both centrally shared and contract sites to send email to a problem account so we can keep better track of problems and make sure we don't forget or lose any requests. Also, we have tried to train the users not to send email to any particular person in our group so that if someone gets sick or leaves for a vacation, the user's request will not sit idle for the duration of their absence. This also minimizes the number of email articles system staff must sort through daily. All problem email on the contract machines also gets copied to the co-administrator, who may choose to reply immediately, or wait for the central group to respond.

The consultant reads the problem email several times daily and insures that at least a minimum response is provided as soon as possible—and no later than a day later. If the consultant cannot answer a question, or if a problem report seems indicative of a serious problem, the email is forwarded to system staff, or a phone call to system staff is made immediately. It is important to note that although problem email is the primary responsibility of one person, it is not answered by only one person. If the first person to read the problem can't supply an answer it will be bounced around for discussion among the whole group, each of us adding bits and pieces of explanations. Users sending email to problem gain access to a pool of information, not just a single point of information. Our co-administrators get copies of the replies so they can learn how certain questions are answered.

Daily Machine Monitoring—reading machine reports.

On our contract machines and our time-sharing machines, we have installed automated programs such as *watcher(1)*[5], *{daily,weekly,monthly}runs*, *rdist(1)* and even backups. We are currently developing a program to check the consistency of key files (or parts thereof) and directories. The project name--*hobgoblin* is from the quote, "Needless consistency is the hobgoblin of little minds" by Emerson. This project is the first of several security and consistency projects by the UCC Unix operating system group. Unlike other consistency checking systems, *hobgoblin* will check for particulars on a file-by-file basis, without chewing up large amounts of disk with checksums of every file for every architecture and operating system revision.

These programs send daily email reports on the status of each machine. Even with a consultant sorting through user problem reports, central system staff get 20 to 40 email articles daily from the automated processes on every machine. All of these reports are not sent to central system staff, but instead are directed to a single email-reading account which we call "tricorder."[6] This has cut down the number of email articles individual central system staff members receive to only five to fifteen email articles daily and allows us to share email reports from all machines. System staff members rotate reading tricorder email daily, with the pager, similar in concept to "system-administrator-of-the-day."

Weekly Maintenance Sessions—doing the work on-site.

In addition to the daily checks by tricorder, each contract is regularly scheduled for weekly maintenance sessions of up to 4 hours long. A list of priorities, or change requests, is maintained by the department programmer and the central programmer to help reduce the potential problem of "jobs waiting forever in the queue." During these sessions, the co-administrator can set the priority of the tasks requested and can

[5] *watcher(1)* is a utility for monitoring machines. See Ingham, Kenneth, "Keeping watch over the flocks by night (and day)", USENIX, 1987.

[6] "A tricorder is a portable sensor/computer/recorder ...carried by any crew member. ...It can be used to sense, analyze, identify and keep records on almost any type of data on planet surfaces. The tricorder is carried by the communications officer to maintain records of what is happening ... or by anyone else who needs a portable scientific tool." Trimble, BJO **Star Trek Concordance**, Ballantine Books, 1976.

always count on central system programmer to be at the department site to work on requests and problems during this time. We use the "ABC" method of prioritizing tasks, where: A means "do it yesterday!"; B means "do it this week, or next."; and C means "this would be nice to have in the future.". Special projects, like new hardware or software installations and upgrades, are scheduled as they arise and aren't restricted to the maintenance session times.

GROUP AND INDIVIDUAL RESPONSIBILITIES

As members of a central group we work closely and usually in teams of two. Our work load is comprised of several layers of tasks. Some of these responsibilities are handled by single people and some of them are shared by the whole group. We have breakfast together once a week to co-ordinate our activities as a group and otherwise meet frequently to plan and discuss changes.

Machine-Specific Tasks— correcting a dial-up line or installing a disk.

These tasks are usually done by the primary system administrator of the machine. If emergencies keep that person busy, they may be done by other group members since we all follow the same procedures when working on any machine. If a better way to do something becomes apparent to one of us, we all review it and agree to follow the new procedure before implementing it anywhere. Any changes to a system are reported to the backup person.

At all times, the two administrators (central sys-staff and department co-administrator) may make decisions about changes to a particular machine. It is the primary responsibility of these two people to insure that a machine is performing properly and that all changes to the system are made properly. All other people who are responsible for parts of the system must work directly with these two people.

Software Subsystem Tasks—maintaining *uucp, news, accounting*, etc.

These tasks are done by one person across all machines. This is one example of how some changes are maintained and initiated under central control. Changes are are distributed from a central source to the same location (usually **/usr/ucc/lib/<subsys>**) on every remote site using *rdist*. The remote sites can then *rdist* the files from there. Large software package installations or upgrades, however, are scheduled by the primary system administrator and follow any special instructions given by that person (or the co-administrator). Because we each have software installations and subsystems that we maintain across all machines, we all keep in direct contact with each system and we are better able to cover for each other.

Projects— beta tests, new operating system release testing, and software development.

These tasks are done individually and in groups. We have found that if we split the machine load evenly between us, so we each have machines and subsystems to maintain and projects to lead, we never get around to the projects. For this reason one programmer will always be given a period of several months to devote to projects. That programmer does not have any machine or subsystem responsibilities to interrupt the project. Some low-priority projects are able to be done by programmers with machine responsibilities.

PROGRAMMER LOAD BALANCING

A central administrator will always be responsible for the three types of tasks outlined above—the balance of one over the other will depend on the particular skills of the programmer. The diversity of the systems ultimately determines how many systems a central administrator can maintain. We assume that a central system programmer can maintain a maximum of five server-client groups or four centrally shared machines.

The current machine-specific responsibilities of one of our current Senior Operating System Programmers includes:

• a graphics lab—six workstations serving 10 to 50 users. This site has a co-administrator to handle the graphics software.

• a centrally shared system for scientific research—an Alliant FX-8 serving over 100 users.

• a department server-client group—six Sun Workstations serving 100 to 200 users. This site has a co-administrator to handle the statistical analysis software.

• a centrally shared system—a Vax 11/750 serving less than 50 active users, but hundreds of people rely on its network services.

• a programming group—four Sun Workstations serving four users. This site has a co-administrator to handle group development software.

This programmer also has primary responsibilities in maintaining and upgrading several large software subsystems on all computers.

CENTRAL LIBRARY FOR SOURCES

Because each department may demand different architecture and software releases, procedures and mechanisms for maintaining source code versions and architectural differences are required. We provide a file system of source codes which each of our contract computers may NFS mount in order to run *make install* or just to browse. When we get new releases to software, we do not distribute them to all contract sites at the same time. Instead, the contract sites are informed of the availability of the new release and the co-administrator may schedule the upgrade during a convenient time for their site.

The source code library is organized by the software's availability to public or licensed users, machine architectures, operating system releases (when critical), and software version numbers. We use the source tree model provided by the X Window System® [7].

CONTRACT OF SERVICES AND RESPONSIBILITIES

A contract for services is preferred over work by the hour since most department programmers know little about system administration and therefore are not often able to determine what tasks to request. In addition, when a novice department programmer has been maintaining a machine for any length of time, we cannot easily predict the amount of time it will take us to perform a requested task. It takes us longer to do the same work on a machine that we do not maintain as it does on a machine that has been under a support contract because of unfamiliarity with, and the abundance of, local changes. Department administrators can become quite alarmed at the bill they are charged for work done under these circumstances. When we establish the fees for our contract services, we estimate the amount of time for upgrades and maintenance to be lower because there should be fewer surprises.

FEE FOR SERVICE

In general, the contract fee is based on the amount of time a system programmer may spend on a system. We base our fee on the number of machine configurations a single person can handle, divided by the University's estimate of what a full-time programmer should cost the University. Our goal is to "break even" and not produce any profit by charging above our costs. Each department requesting a contract can expect to pay for the percentage of a full-time programmer its site will require. Currently, we have departments with contracts of up to 1/4 of the time (and salary) of a system programmer. A contract of this size would require up to 7.5 work hours weekly (We estimate that we can bill up to 75% of a programmer's hours, or (40 hours*.75)/4 for a quarter of a programmer.) and may comprise a server-client group of ten workstations with 50 to 100 users, or it may comprise a mini super computer with 75 users.

We use a time reporting system[8] to verify that we are spending the number of hours we anticipated. The

[7] X Window System® is a trademark of the Massachusetts Institute of Technology.

[8] This system is being developed at the UCC. It is a rather crude system which requires time spent on each contract to be manually recorded by the programmer and then fed into an Ingres® database by a data entry staff. This system is being

contract fees can be adjusted based on this information, if necessary.

SOFTWARE AND OPERATING SYSTEM LIMITATIONS

We have limited our contracts to operating systems that are derivatives of the Berkeley Standard Distribution. This limits the scope of knowledge required by each central system programmer and allows us to work with fewer models. These include Mt. Xinu® [9], Concentrix® [10], and SunOS® [11], among others. We have been asked to consider supporting operating systems based on a version of Unix System V® but have not yet seen enough demand on campus to begin supporting them.

On each machine we install our software into **/usr/ucc**. This directory contains other common directories: **bin, adm, etc, man, lib** and **doc**. Every utility installed in **/usr/ucc** must have a man page in **/usr/ucc/man** and needed libraries in **/usr/ucc/lib**. Anything in these directories is maintained and supported by the UCC. We also set up areas for each of the departments in **/usr/<department>**. We don't really care if they use **/usr/local**, just so they don't *ever* install local software into **/usr/bin, /bin**, or other directories that are part of the operating system distribution. Users are instructed to contact the local programmer for assistance when using software utilities in **/usr/<department>** and to send email to problem if they have any other problems or concerns with the system.

CONSISTENT RESPONSIBILITIES

The hardest part of sharing responsibility for a machine is determining who will do what. Some responsibilities are shared by both the central administrator and the department co-administrator. These are:

- use *RCS(1)* to record any changes to ascii system files.

- follow the Super-User policy (outlined below).

- keep the system log book up to date.

- maintain a list of requested tasks or problems with priority levels associated with them.

Central System Programmer Responsibilities

- maintain and install software into operation system areas, including operating system upgrades or patches. Alterations to the operating system may be performed, provided they fall within a negotiable range.

- update the **/etc/hosts** and **/etc/networks** table from the Network Information Center, and provide and maintain a flexible sendmail configuration file.

- maintain and install centrally supported software in **/usr/ucc** at the request of the co-administrator. This software includes packages which are not part of the operating system distribution and may include public domain software and licensed products. The licensed products must be paid for under a university site license or by individual department licenses.

- maintain and install vendor-supported software or device drivers. We will act as intermediary to a support organization if bugs are suspected but we won't provide program support for bugs even if the organization that should be providing support does not. We will not write device drivers, special accounting programs, or applications programs. These services can be contracted at a fee for service rate.

used by all UCC personnel (programmers, consultants and administrators).

[9] Mt. Xinu® is a product of Mt. Xinu Incorporated.

[10] Concentrix® is a trademark of Alliant Computer Systems Corporation.

[11] SunOS® is a trademark of Sun Microsystems, Incorporated.

• assist in trouble-shooting hardware failures.

• monitor the system via *watcher* and *{daily,weekly,monthly}run*. We will review utilization and error logs to help maintain reliability of the system and suggest performance improvements.

• schedule all changes by other central system staff with co-administrator.

• give advice to the co-administrator on department software. Help the co-administrator plan new hardware configurations and make purchases.

• provide backup scripts and recommend a backup schedule.

• perform disk re-partitioning and filesystem re-construction as required, including the addition of new clients on a server.

Department Co-Administrator Responsibilities

• maintain department software in **/usr/<department>** and support the department users. This would include any "home-grown" software.

• install accounts (access to system files **/usr/lib/aliases, /etc/passwd, /etc/group**).

• determine the direction of growth of the facility.

• respond first to trouble calls by being first on the on-call list. This list may be posted in the user area at that site. At the first sign of hardware or system trouble, contact a hardware technician or system staff member and provide access to machine areas. Each department should have contracted for hardware support from a support group. The University of Rochester has a Computer Equipment Services department which has trained technicians for maintaining a wide variety of equipment on campus.

• act as, or work with, the local operator by performing backups and printer output maintenance.

SUPER-USER POLICY

The root password is not distributed to all people with root, or super-user, privileges. Logging into a system as root is strictly forbidden and is monitored by the operators who have been instructed to kill a root login shell if they cannot verify its legitimacy. Access to privileged commands is given on a machine by machine basis. If a user has been given super-user privilege, they may use the command *suex(8)*.[12] *suex* gives the user a root shell from which to execute desired commands. It may be used to execute single commands by prefacing the command with *suex*, similar to the usage of the command *nice(1)*. Accounts with super-user access must be as well protected as the root account. Although this program creates many more ways to break into the root account, we find its convenience out-weighs its security risks. Because no one logs in as root to perform system maintenance, we are able to use system accounting records (via *lastcomm(1)*) to audit suspicious activities by any person having super-user privileges.

Below are some guidelines we follow:

• never make system changes as root. System staff and co-administrators have permission to write in most directories via membership in group sys, wheel, or the equivalent.

[12] *sux(8)* (super-user execute) was written by Jim Mayer at the University of Rochester, July 1983. We re-named it *suex*, locally.

• never allow the use of your account by another person than yourself. Never leave your account logged in on a terminal or an unlocked workstation.

• never operate continuously under *suex*. If a task can be performed without super-user privilege, it should be performed under your own account.

• only make changes to what you agreed on in advance. Even when changes seem trivial and you are knowledgeable about a certain subsystem (*uucp*, for example), it is best to allow all changes to be maintained by the person with primary responsibility for that subsystem.

PROCEDURE FOR ADDITIONAL CHANGES

This procedure must be followed by any programmer, or person with super-user privilege, who wishes to make system changes which are not part of their regular responsibilities. We have found that this procedure has enabled people to learn system administration tasks quickly and easily, and facilitates troubleshooting when things go wrong. Our co-administrators follow this procedure to make changes to the operating system, such as installing a new kernel.

1. **make the request.**

   A programmer notifies central system staff of a desired change and a proposed schedule. A detailed outline of the entire procedure must be provided at the command level. If necessary, central system staff may provide or help prepare the commands required for the desired change.

2. **schedule the change.**

   Central system staff will verify the risk and impact of the change and help schedule it during the best time-frame for that system. Modifications to the steps outlined in the programmer's directions will be made at this time. The programmer is not to proceed until system staff have approved the change. If the risk or impact is high, the change must be made by central system staff or under their direct supervision.

3. **monitor the system.**

   System staff will be responsible for verifying that the changes are made properly and for monitoring the system during and after the change period. System staff will also make sure that notes were made in the system log book for the changes.

4. **report the status.**

   The programmer making the changes must notify system staff of the success or failure of the task at the end of the time-frame. If the change did not take place during the scheduled time, it's important to send out notifications that the change did not take place, since some people may otherwise assume that the change was made. If the change was unsuccessful, system staff will review the procedure and suggest alternative approaches. Usually, if the programmer gets into trouble and system staff aren't assisting directly in the work, system staff will be paged.

SUMMARY

The UCC has demonstrated at the University of Rochester how a central group can maintain machines running a version of the Unix operating system without losing distributed control. We have evolved into an environment of distributed shared resources. We are now starting to maintain contracts for centrally maintained file servers with diskless workstations owned by different departments and maintained by department co-administrators. In addition to our centrally shared computers and our department contracts, we will continue to provide shared resources for university site licenses, USENET news, UUCP, and per-hour consulting.

WHERE OUR APPROACH WILL FAIL

There are circumstances under which our model will fail, and circumstances where we have not tried to apply it at the University of Rochester:

- If a department has enough funds, it will hire a programmer, despite any of the benefits of a central group. Usually the case is the reverse and departments do not have enough funds and must pool their funds to share resources with other departments. Other departments may never realize the amount of effort required to maintain machines, and will never believe that the benefits are worth the costs. These departments usually assume that computers, especially small ones, will run by themselves.

- If the department has a critical application and must get immediate attention when problems arise it must have an in-house programmer to respond immediately. In this case, the department may develop a low tolerance for waiting in a queue for anything.

- If the department is doing research at a technical level and must interact closely with the operating system, it must have a system programmer in the department to assist in the development.

We believe that our model will continue to succeed when a department contract grows to machine numbers of 40 to 100. What is time-consuming in maintaining the department site is not the number of machines (although because of the hardware failures especially, they will always affect the maintenance load), but the number of models which must be maintained. A department may have 40 machines, but only two master copies, or models. Within a department, a single master and many duplicate machines are easily accepted.

WHY OUR APPROACH WILL REMAIN ATTRACTIVE

A shared central group of operating system programmers provides several advantages to a university: The university as a whole saves money by not hiring programmers for every department. The total cost to the department with a contract is lower because the productive time of a programmer who is already familiar with a system is higher, therefore the billable hours are lower. The programmers benefit by being in a support group of other programmers who share ideas and information on a daily basis. The central programmers are not asked to be a "jack of all trades" and may specialize and receive more training in areas of interest.

The additional services our central operating system group has provided the University of Rochester include:

- a single group to do evaluation and testing of products of interest to the entire university.

- a centralized programming team to do development projects which all departments can easily benefit from.

- a system administration training class for co-administrators.

- a vehicle for communicating between departments, to help service common needs within all departments, and to identify software packages which can be purchased and shared by many departments.

ACKNOWLEDGEMENTS

Many thanks to all of the UCC co-administrators who contributed better explanations of what services we provide than we were aware of, and to the UCC Unix operating systems group, who really make these contracts work—despite whatever I said here. Special thanks to Carolyn Wasikowski at the Laboratory for Laser Energetics for helping to improve the clarity of the writing.

OLC: An On-Line Consulting System for UNIX†

Thomas J. Coppeto
Beth L. Anderson
Daniel E. Geer, Jr.

Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
{tjcoppet,beth,geer}@ATHENA.MIT.EDU

G. Winfield Treese

Cambridge Research Laboratory
Digital Equipment Corporation
Cambridge, MA 02139
treese@CRL.DEC.COM

ABSTRACT

Helping users learn the intricacies of UNIX, particularly in a custom environment, is always a challenge. Helping thousands of users in an environment that is distributed both geographically and computationally is especially difficult. Project Athena has developed an "On-Line Consulting" system (OLC) that enables users to ask questions of consultants located "somewhere on the network." OLC allows a staff on the order of twenty students to handle the questions and problems of over 8000 users on a network of more than 900 workstations. This paper describes the motives and design goals for OLC, its implementation, and some of the results of its three years of operation.

1. Background on Project Athena

Project Athena was originally conceived as a five-year experiment in the uses of computers in undergraduate education at the Massachusetts Institute of Technology. With support from Digital Equipment Corporation and International Business Machines Corp., M.I.T. embarked on the construction of a distributed computing environment built on scientific/engineering workstations (VAXstation II and RT/PC). In addition to this environment, Athena has funded over one hundred projects for developing educational software for use in the M.I.T. curriculum. In 1988, Project Athena was extended by M.I.T, Digital, and IBM for three years. An overview of the Project can be found in ref[1].

The Project Athena computing environment is built on a network services model of computation. Virtually every resource used during a login session is supplied by a remote server: software, personal files, mail, message-of-the-day, printing, *etc*. Athena currently has installed over eight hundred workstations and seventy server machines yet the operations staff numbers approximately eight. A central requirement of all services is that they permit easy upward growth in numbers. The Athena model of computation and

† UNIX is a trademark of Bell Laboratories.

building such a distributed environment is described more fully in a series of papers presented at the Winter 1988 USENIX Conference.

The computing experience of Athena users covers a wide spectrum, from those who have never used a computer to advanced UNIX hackers. Even experienced UNIX users often have questions, especially since Athena has augmented the basic system with several additional software packages. Currently, there are approximately 10,000 registered users of Project Athena.

2. The Problem of Consulting

Each September, more than 1,000 new undergraduates arrive at M.I.T. Many of them are eager to start using Athena workstations at once while others will wait until required to do so by enrolling in a class that uses the Athena system. In either case most students must learn to use the computing environment during their study at M.I.T.

UNIX has never been regarded as an easy system to learn, and the process is complicated by the network server configuration and the addition of other software packages (*e.g.*, the X Window System.) This problem was recognized early in the Project, and a consulting staff was established to assist users by answering their questions about the system. Most consultants have been undergraduates who work for Athena part-time. In addition to the paid consultants, many students have volunteered their time to assist other users with the use of the Athena system. These OLC volunteers were originally Athena staff and members of the Student Information Processing Board (SIPB) at M.I.T., but the group has expanded to include many other users.

>From the beginning, it has been impossible to provide consulting coverage in all locations twenty-four hours a day, seven days a week. In 1984 the need to provide some kind of on-line support for consulting became clear and work on OLC began.

3. The On-Line Consulting System

OLC is designed to be the rendezvous service between users and user consultants. Transactions between users and consultants are immediate. If there are no consultants available at the time a user asks a question, the question is held until the question is answered, even if the user logs out. A user may be helped by only one consultant at a time, however, her question may be passed from consultant to consultant or remain in a queue until the question is resolved. OLC is also designed to take advantage of other services in the Athena service environment in a way that does not compromise the basic reliability constraint.

The complete OLC system consists of three pieces: a user program *(olc)*, a consultant's response program *(olcr)*, and the coordinating daemon *(olcd)*. In addition to these programs, there is a collection of "stock answers" or prepared responses to common questions. Athena has also established certain management procedures for using OLC which are described later in this paper.

3.1. Design Goals

Some aspects of the design of OLC are those common to the design of any network service. OLC must be:

Scalable:	One of the goals of Project Athena is to provide a large-scale computing environment with minimal operational overhead. OLC must be able to quickly process and maintain large numbers of requests and to meet increased demand gracefully.
Reliable:	The service should be available whenever any user requires it (all the time).
Recoverable:	If the system crashes, it should recover with minimal human intervention, preferably with none at all. OLC should restore its state such that all current conversations are preserved.
Usable:	In some ways, OLC provides the equivalent of a "help" system. When a user understands nothing else, he should be able to easily use OLC to contact a consultant without first asking how to use the program.
Inexpensive:	As the user population grows, it is inevitable that OLC is to become a service provided by users, for users. OLC must therefore be simple to manage and maintain.

3.2. The OLC Daemon

This daemon provides a central rendezvous for user questions such that consultants can read and answer them. All user-consultant transactions are mediated by the daemon so it can handle proper notification, logging, and scheduling of the questions and transactions.

3.2.1. Connections

The daemon maintains the connections between users and consultants. A user may be connected to only one consultant and vice-versa. Connections are merely symbolic pointers within the daemon. OLC clients do not interact directly with each other. This has the advantage of maintaining reliable, current logs of all transactions and state data.

3.2.2. Types

Three different classifications of users are currently defined in the OLC daemon:

1. User who is asking a question
2. Staff consultant
3. OLC volunteer

Any user, including registered consultants, may ask a question. Active questions entered by consultants do not interfere with their role as consultants. Consultants come in two flavors but their functions are identical. The distinction is made merely to allow staff consultants extra flexibility within the system (see below). A consultant is registered in the OLC database by the OLC manager.

3.2.3. Specialties & Priorities (Auto-connection)

Users are best served when their questions are tended to promptly. Consultants may *sign on* in OLC so that the daemon will automatically connect them to waiting users. The daemon will connect the consultant to the first user in the queue whose question topic matches one of the consultant's declared specialties. A specialty is a topic in which the consultant has a strong background, indicating a routing preference for questions. The specialty is declared in the OLC database by the OLC manager.

The daemon will connect new questions with consultants currently signed on. This is determined by the level at which the consultant signs on and by his specialty list. A staff consultant may sign on at one of three levels (in order of priority):

on duty

on

on urgent

The duty level is the front line; consultants signed on at this level are connected to users ahead of any other consultants signed on. The urgent level is useful when someone wishes to help users, but only if no other consultants are available.

The OLC volunteer may only sign on one level. If there is more than one consultant on a given level, a new question is given to one of those consultants with a specialty in that question topic. In addition, volunteers will not automatically connect to questions with topics outside of their specialty list. This feature allows for volunteers who are knowledgeable in a few areas of the system, yet may know little about other areas.

This design not only serves users promptly, but makes technical matches between users and consultants.

3.2.4. Logging

All transactions between a user and a consultant are stored in a log which resides on the server. This feature is necessary so that questions can be transferred among consultants.

Upon the resolution of a question the daemon forwards the conversation log to a third party archive service. The logs are categorized by the topic of the question asked. The daemon is configured to work with the *notes* conferencing system developed at the University of Illinois (ref[3]), or the *discuss* networked conferencing system developed by the Student Information Processing Board at M.I.T. (ref[4]). These logs are kept as an official record of all transactions over OLC. Often a consultant will need to refer to a past conversation for reference or another consultant may scan the logs for accuracy and polite interaction. The OLC logs are a useful mechanism for Project Athena to evaluate itself by noting common problems and questions about the system.

The consulting staff uses this information to prepare itself for commonly asked questions by creating stock answers (see section 3.5). The logs are also a useful pulse of the community.

3.2.5. Notification

The daemon will notify users and consultants of any status changes to their conversation such as a new connection or message (See section 4.4).

```
% olc
Welcome to OLC, Project Athena's On-Line Consulting system.
Copyright (c) 1988 by the Massachusetts Institute of Technology.

If you need help with this program, type 'help' at the "olc>" prompt.
There are 2 busy consultants, with 4 waiting users.

There is still one machine in building 4 down due to hardware problems.
This should not affect most people, although some workstations in
buildings 16, 4, and 2 may not work correctly. Repairs may extend over
the weekend.
                -- 8:00PM, September 30

Please type a one-word topic for your question.  Type ? for a list of
available topics or ^D to exit.

Topic: workstations

Please enter your question. End with a ^D or '.' on a line by itself.
How do I get a clock on my screen?
^D

There is no specialist currently available for topic 'workstations'. Your
request will be forwarded to the first available consultant.
olc>
```

Figure 1: The olc startup screen (with typical contents)

3.2.6. Host Requirements

The OLC daemon runs on a private secure host to avoid tampering with user conversations or logs. The host must be accessible via TCP/IP connections within the distributed environment. In addition it must have enough accessible disk space to house the active conversations and the resolved logs.

3.3. User Client: *olc*

3.3.1. The startup screen

When a user starts olc, the client displays three pieces of information (see Figure 1):

1. How to get help within the program: every command may be explained by 'help <command>'. These explanations are straight forward and simple to avoid generating additional questions.

2. The status of the OLC queue: the queue itself is hidden from the user. However users are generally interested in knowing about how long it may be until their questions are seen by a consultant. This status provides the number of consultants and the number of active users in the queue. The user may request this information during an OLC session.

3. A short message of the day: Occasionally an event may occur within the system which causes many users to ask the same question. To ease the load on consultants in such an event, *olc* displays a short message which may account for down systems, etc. This message can be used to display OLC down times, system problems, consulting hours, or even new ways to get help on the system.

3.3.2. Use of olc

After *olc* displays the message of the day, the user is then asked to supply a topic from a predefined set and the text of his question. This is all that the user is required to do until she is connected to a consultant. During the time the user waits, she may send additional messages which will be held for the consultants in the user's log, query the status of the queue, or exit *olc* and do other computing in the meantime. When the user re-enters *olc,* the client will skip the initial query for a topic and question and place the user at the OLC prompt. At this point the user may send new messages or replay the conversation so far.

The two commands the user must know are *show* and *send. Send* will take additional input from the user which will be appended to the log on the server. If the user is connected to a consultant, then the server will notify the consultant of the user's new message. *Show* will display new messages from the consultant (see figure 4 for an example).

3.4. Consultant reply program: olcr

Olcr is essentially *olc* with additional commands to manipulate connections with users. Consultants have the ability to connect to users, send messages to the connected user, forward or resolve the user questions (see *Figure 5* for an example).

3.4.1. The OLC queue

The OLC queue seen by the consultant is shown in *Figure 2.* The list contains all the essential information in OLC. The two left columns display the user name, location, and status. The status may be one of *unseen, pending, active, logout.* The next column lists the consultant's username. A consultant listed on the same line as the user indicates that he is connected to that user. The last columns display the number of consultants who have been connected to that user and the topic of his question. It is possible to configure the daemon so that it will forward the question to a log when a certain number of consultants have been connected to the question but have failed to resolve it. This feature prevents the ever increasing size of the queue and bouncing of unanswerable or difficult questions inside the queue. The OLC Manager (see *section 5*) can take care these forwarded logs at a later time.

3.4.2. When a user logs out

Olcr has the ability to send mail to a user in the event the user should log out while having a pending question in the queue. This feature enables consultants to eliminate already answered questions in the queue and allows the user to see consultant responses.

In the Athena system, Mail is forwarded to a central mail hub. The Athena mail hub is a separate service which delivers mail to the appropriate address. The olcr client will notify the consultant if the user is unknown to this service. Specific user addresses may also be specified for use with more traditional systems.

3.5. Stock Answers

After OLC was first put into service, it was discovered that users frequently asked the same, or very similar, questions. For example, "How can I print my paper?" is a common question. There were also a number of less common but somewhat technical questions, such as "How can I link my C program with a FORTRAN library?" It was tedious for consultants to retype answers to common questions, and difficult to write answers quickly for more technical topics. Over time, consultants began to keep individual caches of answers, and occasionally these caches would be shared with other consultants.

Once the problem was recognized, a solution was obvious. The consultants gathered a collection of these "stock answers," and a simple browser program was written to enable consultants to find an appropriate stock answer quickly. Once found, the answer could be tailored to the specific question and sent to the user.

Stock answers are organized in a hierarchy, with the categories and placement of answers determined by the OLC manager. The browser interface is similar to that of *notes* (see *Figure 3*).

Single-letter commands are executed immediately with no carriage return required; an entry is selected by typing its number as displayed in a menu on the screen. The browser has full support for saving entries to files and for adding new entries. Some experience with the interface is required to use it easily, but it works very quickly once it has been mastered. Currently, the browser is also available to users as a supplementary set of on-line documentation.

```
      olcr> list

User                           Status      Consultant                   Topic
hyland@M66-080-2.MIT.EDU       (pending)                                (2) emacs
jamarroq@HAWAII.MIT.EDU        (logout)    geer@E40-342F-2.MIT.EDU      (5) fortran
mcampos@GRILL.MIT.EDU          (logout)    beth@KLEE.MIT.EDU            (2) xwindowsystem
gmbelaus@BECKS.MIT.EDU         (logout)                                 (1) scribe
twleung@E40-358D-1.MIT.EDU     (logout)                                 (1) mail
enprahba@RENOIR.MIT.EDU        (pending)                                (3) unix
annette@GRUMPY.MIT.EDU         (active)    elsj@M4-035-10.MIT.EDU       (1) other
carla@SNOOPY.MIT.EDU           (active)    vanharen@FRIES.MIT.EDU       (1) accounts
tjcoppet@PICASSO.MIT.EDU       (pending)                                (1) lisp
not connected                              treese@CIROCCO.MIT.EDU
not connected                              tjcoppet@PICASSO.MIT.EDU

olcr>
```

Figure 2: The OLC queue as displayed in olcr

Currently, an interface for the X Window System (ref [5]) is under development. Advantages of this version include the ability to view a directory of entries and a particular entry simultaneously as well as a "friendlier" method of finding desired entries. It is possible that a more general "help" system can be based on this interface.

4. Interaction with Other Athena Services

4.1. Athena mail hub

To control the flow of mail within the Athena network, as well as in and out of Athena, all mail is sent to a central mail server which has the information to forward the mail to the appropriate location. These locations are known as Post Offices, ref[6], which are essentially mail holding servers. The user can use any one of the mail clients available on Athena to retrieve and read their mail messages without knowing about the existence of these servers.

4.2. Hesiod

Hesiod, ref[7], provides name service for Project Athena. The clients query the Hesiod service for the location of the OLC server. This location may be changed at any time without altering the any of the OLC programs.

4.3. Kerberos

In an environment of more than 800 public workstations on an insecure network it is necessary to confirm the identity of the consultant when he is assisting users within the OLC system. Services at Project Athena use a third party service called Kerberos, ref[8], to perform this authentication. As a policy, the OLC daemon refuses connections from consultants who fail this authentication mechanism. Currently no authentication is required to ask a question because a user may have a question about this authentication system and turn to OLC to find the answer.

4.4. Zephyr

The OLC daemon sends all notifications directly to the user. Normally this would occur via a system *write* message to the user's terminal or workstation, however, this requires the OLC daemon know which machine the user is logged into at all times. This is not possible when a user logs out and logs in again at another machine, and does not make any requests from olc which would update his location. To solve the problem of finding a user on the network, Project Athena has developed a location and notification service known as *Zephyr*, ref[9]. The OLC daemon sends notifications via this service, and, by default, the user will receive the notification at the tty or inside a *windowgram* which is essentially a small window containing the notice OLC has sent. This is however user configurable and if this system fails, the user will receive the message directly from the UNIX write function.

```
                    On-Line Consulting Browser
            /usr/athena/lib/olc/stock/stock_answers/emacs


            1  Controlling the emacs window size
            2  Using the mouse buttons in Emacs
            3  Moving the cursor without the mouse
            4  Finding what line you are on
            5  How to set a mark
            6  Line number mode in Emacs
            7  How to set auto-fill by default
            8  How to recover Emacs autosave files
            9  Customizing Emacs using a .emacs file
            10  How to use Emacs with scheme
            11  How to use Emacs with a newer scheme
            12  Using TAGS in Emacs
            13  Rebound keys for VT240

                       ** End of Index **

    olc_browser>
```

Figure 3: The OLC Stock Answer browser (a typical display)

4.5. Network dependencies

The first requirement for a functional OLC system is that the OLC server be operational. If the OLC daemon is running on a working machine the clients simply need to establish a TCP/IP connection. Generally if the user can successfully log into a workstation, he can successfully use OLC. The only third party service requirements are *Hesiod* for the name service, *Kerberos* for authentication, and of course, the user must be able to obtain the client. At Athena, the client is delivered via a cluster library server which provides most of the essential software to workstations in a particular location. If any one of these should fail, the workstation is unusable.

Interaction with other services like *Zephyr* is non-essential since the daemon is capable of sending notifications directly to the user's workstation.

5. Managing OLC

5.1. OLC database

The database resides on the daemon host. It contains a list of usernames allowed to act as consultants along with the topics in which they are specialists. The usernames may be marked to indicate if the consultant is a volunteer. This file is stored in ASCII text so it may be edited easily.

The OLC topics are listed in a separate file and also stored in ASCII text. Since the daemon will file conversations by topic, it is necessary that the logging mechanism be updated for any changes to this topic file. Either file may be edited without halting the daemon.

5.2. The OLC queue

Some difficult questions may linger in the queue, especially if the user logs out since logged out users will not be automatically connected to a consultant. It is advantageous to keep the OLC queue short. Large queues often contain many questions from users who logged out and did not provide sufficient information so that a consultant could send a definite answer.

In general, every question is answerable by at least one member of the Athena staff. It is ultimately the job of the OLC manager to find the answer to difficult questions. In rare cases, such as requests about network addresses of remote hosts or debugging of large programs, the user is given information on how to help himself. Often this is a reference to materials off-line.

5.3. Orientation

The OLC manager is also responsible for orienting new consultants and volunteers with the proper use of the *olcr* client. An olcr manual exists but orientation is geared toward handling difficult situations with users that might lead to misunderstandings or arguments. Since OLC is one of the major user interfaces to Project Athena, it is necessary to ensure smooth communication between users and consultants.

6. Problems and Future Directions

Soon OLC will allow consultants to answer multiple questions simultaneously. The advantage is that other users may be served while a consultant is waiting for one user to respond, i.e. consultants will be more efficiently utilized.

As more OLC volunteers are added, OLC approaches the model of a system of users consultation provided by users. These volunteers are largely responsible for the success of OLC. The number of volunteers will gradually increase to fill the need.

An invaluable source of information about the functionality of the system and consultant performance can be found in the OLC logs. However, the man-power required to read over 6000 questions each term is greater than what Athena is able to provide.

The limits of OLC scale are dependent upon the limits of the OLC server (an estimated 300 questions). With some work, the OLC system can be expanded to utilize multiple servers simultaneously. Each OLC server may be responsible for some area of specialty or area of administration. The limits on the type of scale is dependent upon the limits of Athena's authentication, notification, and name services across internet regions.

7. Conclusion

A team of 10 consultants and 20 volunteers provide help to users 24 hours each day. Even when a particular consultant may not know the answer to a question or problem, it is reassuring to the user to know that he is able to contact another person on the network. OLC has proven to be the solution for providing user assistance on a broadly distributed network system.

8. Acknowledgments

The original design of Project Athena's OLC (based on an on line consulting system provided by Information Services at M.I.T.) was done by Bill Saphir. Dan Morgan and G. Winfield Treese of Project Athena continued the implementation through to a usable system.

We would also like to thank Marc Campos, Theodore Leung, and Gregory Belaus for maintaining and providing new ideas for the OLC system; Carla Fermann, Chris VanHaren, and Linda Kim for reviewing early drafts of this paper.

9. Sample OLC sessions

The following pages contain sample *olc* and *olcr* sessions. The text contained in the boxes are asynchronous notifications sent by the daemon via *Zephyr* or */bin/write*.

(continued from Figure 1)

```
Message from OLC daemon...
You are connected to consultant Tom Coppeto (tjcoppet@PICASSO.MIT.EDU)
Tom has received your question.
```

```
Message from OLC daemon...
Tom has sent a reply.
To read it, use the 'show' command from olc.
```

```
olc> show
time: 88/09/28 03:07:23

Hi Joe,

To get a clock on your screen type:
        xclock &
When your cursor changes to an upside down L, click the left mouse button
where you want the clock to appear.

                        - Tom

olc> send

Enter your message. End with a ^D or '.' on a line by itself.
Thank you.
-Joe
^D

What now? send
Message sent.

olc> done

Using this command means that the consultant has satisfactorily answered
your question. If this is not the case, you can exit using the 'quit' command,
and OLC will save your question until a consultant can answer it. If you
wish to withdraw your question, use the 'cancel' command.
Really done? [y/n] y

Goodbye from OLC. Glad to be of service!
%
```

```
Message from OLC daemon...
Tom has marked your question resolved.
```

Figure 4: An example olc session with joe user.

```
Message from OLC daemon on MATISSE.MIT.EDU...
You are connected to user Joe User (jruser@SOMEWHERE.MIT.EDU)

        olcr> replay

Log Initiated for user Joe User (jruser@SOMEWHERE.MIT.EDU)
      (88/09/28 03:14:45)

Topic:       workstations

Question:
How do a get a clock on my screen?
____________________________________________________________

--- Question grabbed by consultant tjcoppet@PICASSO.MIT.EDU.
    [88/09/28 03:15:03]

--- Connected to consultant tjcoppet@PICASSO.MIT.EDU
    [88/09/28 03:15:04]

olcr> chtopic xwindowsystem
Topic changed to 'xwindowsystem'

olcr> send
Enter your message. End with a ^D or '.' on a line by itself.
Hi Joe,

To get a clock on your screen type:
            xclock &
 When your cursor changes to an upside down L, click the left mouse button
 where you want the clock to appear.

                              - Tom
^D

What now? send
Message sent.
olcr>
 Message from OLC daemon...
 OLC message sent from user.

olcr> show

time: 88/09/28 03:08:46
Thank you.
-Joe
olcr>

 Message from OLC daemon...
 User is done with question.

olcr> done
Enter a title for this conversation: how to get an xclock
Question resolved.
olcr>
```

Figure 5: Example olcr session

10. References

1. E. Balkovich, S.R. Lerman, and R.P. Parmelee, "Computing in Higher Education: The Athena Experience," *Communications of the ACM* **28** (11), pp 1214-1224, ACM (November 1985).

2. G.W. Treese, "Berkeley Unix on 1000 Workstations: Athena Changes to 4.3BSD," pp. 175-182 in *USENIX Conference Proceedings,* Dallas, Texas (February 1988).

3 R.B. Essick IV and R. Kolstad, "Notesfile Reference Manual," *UNIX User's Supplementary Documents,* February 1983.

4. K. Raeburn, J. Rochlis, W. Sommerfeld, and S. Zanarotti, *Discuss:* An Electronic Conferencing System for a Distributed Computing Environment," in *USENIX Association Winter Conference 1989 Proceedings,* February 1989.

5. R.W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics* **5** (2), pp. 79-109 (April 1987).

6 M.T. Rose, "Post Office Protocol (revised)," University of Delaware, 1985.

7. Stephen P. Dyer, "The *Hesiod* name server," In *USENIX Association Winter Conference 1988 Proceedings,* pp. 183-190, February 1988.

8. J.G. Steiner, B.C. Neuman, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," pp. 191-202 in *USENIX Conference Proceedings* Dallas, Texas (February 1988).

9. C. Anthony DellaFera, "The *Zephyr* notification service," In *USENIX Association Winter Conference 1988 Proceedings,* pp. 213-220, February 1988. *Project Athena Technical Plan: Section E.4.1: Zephyr Notification Service,* M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987).

A Distributed Resource Allocator for UNIX® Systems

Griffith G. Smith, Jr.
ggs@ulysses.att.com

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

UNIX®[1] System users have had to tolerate insecure access to nonsharable resources such as cartridge units, tape drives and printers. As commercial applications for UNIX System software become common, more security will be a critical requirement. Software vendors price or restrict the number of concurrent users of their licensed software. These license terms are often difficult to enforce. A resource allocator can address both problems. This paper describes a working prototype of a distributed resource allocator suitable for systems connected by a network. Its design is unusual in that there is no single place where a resource allocation database is maintained. Instead, servers keep information that is of local interest. Clients set up network connections to groups of servers and then negotiate among the servers for the right to allocate resources. The servers dynamically reconfigure themselves into negotiation groups; allocation requests within a group are queued, requests to unrelated groups can be processed in parallel. For a single system, a robust resource allocator is simple to implement, but design of an allocator for a cluster of systems connected by a network is difficult; there are many opportunities for partial failure and incorrect behavior. The allocator achieves a significant level of fault tolerance by exploiting the redundancy of the information stored on multiple servers and by avoiding architecture that is vulnerable to faults. Negotiation protocols combine the distributed information and ensure that a 'correct' decision will be made even when some of the servers fail or have inconsistent status.

1. Introduction

Users and administrators of the UNIX Operating System have typically had a cavalier attitude about accessing nonsharable resources such as cartridge units, tape drives or printers. Access permissions for the device files often allow unlimited access to anyone. For distributed processors with many nonsharable resources, possibly including multi-ported devices, such informality is dangerous; there are too many opportunities for confusion and miscommunication. A resource allocation system is a solution to this potentially disastrous situation. The allocator should let users of systems on a network access nonsharable resources from a common pool without encountering conflicts or deadlocks.

2. Prior Art

Finding existing UNIX System resource allocation software is difficult; browsing through the UNIX System V or 4.3BSD manual pages reveals nothing. A search through documentation of UNIX System dialects offered by other vendors finds scattered programs, usually special-case. The implementors only address the problem at hand: a tape allocation system[1] is buried within a command that passes tape mount requests to an operator; a software license allocator[2] knows nothing about controlling access to magnetic tape drives or printers. Neither allocator is capable of dealing with the case where a user needs two software licenses and a tape drive and must wait until all three are available.

1. UNIX is a registered trademark of AT&T

A '*netnews* special' written in 1983[3] comes much closer to the mark. This simple pair of programs performs two functions:

- An 'allocator' maintains a set of lock files that indicate whether corresponding resources are available or allocated. A resource is allocated by creating a lock file and changing the ownership of all files that permit access to the resource. The allocator can be given a list of resource names; it will assign the ones that are free and return the number of busy resources in the exit code.
- A 'deallocator' deletes the lock files and gives the device files to the *root* account.

This approach is remarkably effective. It is sufficient for the common case where a client asks for one single-ported resource and does not intend to ask for another one. It fails for more complex cases:

- If each of two clients asks for two resources at the same time, they may each get one of them. They will wait indefinitely if they don't coordinate their efforts to get the 'other' resource.
- The two sides of a dual-ported device will be allocated independently; there is no communication among allocators running on separate systems.
- The allocator does not queue requests; one must keep asking for a resource until it is available. The winner is the first person who guesses correctly that a resource has become available.

Despite these flaws, this allocator can serve as a model of an appropriate allocation tool. It has a simple external specification with few options, it is suitable for use as a component of a larger system, and it does one thing well (most of the time).

3. Some Biased Opinions

Building on the above observations (and mixing in some personal bias) I will claim that a well designed resource allocator should satisfy the following requirements:

- The allocator should control access to generic resources, not specifically to tape drives, laser printers, software licenses, etc.
- Access to resources should be granted only by the allocator. Busy resources should be inaccessible to other users; unallocated resources should be inaccessible until they are allocated.
- The allocator should avoid deadlocks when several clients request the same resources simultaneously.
- It should be possible to allocate resources for use across network connections. This should also not be prone to deadlock when two systems ask for the same resources.
- All access points to a resource should be protected. For example, a dual-ported tape drive should have its other port marked 'busy' when access to a port is granted.
- Allocation requests should be queued if necessary. The queuing mechanism should be fair; a client should not have to wait for a resource if the resource is available and no prior requests for it can be granted.

One must also consider the consequences of system faults. An allocator running in isolation on a single system has little need to recover from faults. If the system crashes, the allocations can be abandoned. An allocator that must protect both ports of a dual-ported device has a harder job; an allocation must be honored unless the systems on both ports have crashed. I believe that a distributed resource allocator should have at least the following fault-tolerant features:

- There should be no single system designated as the allocator for the network; if that system were to fail, allocation for the network would be disabled.
- The allocator should tolerate common faults, such as system crashes, without denying access to resources on non-faulting systems.

- A fault should not allow a resource to be accessed improperly.
- A fault should not cause a resource to become permanently inaccessible.
- Fault recovery procedures should be simple. Non-optimal but correct behavior is better than invalid complex behavior.
- The effects of a system fault should be local, if not invisible.

4. *Fault-tolerant Design Alternatives*

I will describe three broad classes of strategies for dealing with system faults. The first choice seems simple: assume nothing will go wrong. This can be called *fault-intolerant* design. Another choice addresses faults directly by adding redundancy and using masking or recovery procedures for anticipated faults. I will call this an *add-on fault-tolerant* design. The third approach is more subtle; the goals are to exploit inherent redundancy and minimize vulnerability to faults by avoiding fault-sensitive architecture, thereby achieving *intrinsic fault tolerance.* Consider the following hypothetical implementations of a distributed resource allocator.

4.1 *Fault-intolerant Design*

A *fault-intolerant* allocator might use a central server that keeps records of all free and active resources in an allocation database. All clients on the network send requests to the server. When granting an allocation request the server uses remote procedure calls or remote command execution to change the status of the access files for the allocated resources on the appropriate systems. This approach has some serious flaws:

- *There is a single point of failure.* If the server fails (program failure or system crash), allocation is disabled for all systems on the network. This can be especially vexing for the common case where the resource you want is on your own system.
- *The design depends on full network connectivity.* A request from system *A* to allocate a resource on the same system may fail because the network path to the allocation server on system *B* is broken. Even when the connection attempt succeeds, the server may be unable to execute a remote procedure on the requesting system.
- *The server becomes a bottleneck.* For the common case where client *X* wants a resource owned by system *A*, and client *Y* wants a resource on system *B*, both clients must take their turn negotiating with the central server, possibly on system *C*. Without a central server, both negotiations could proceed in parallel.
- *Fault recovery is difficult.* If the server has the only copy of the allocation database, a system crash may destroy it or make it inaccessible. Even when the database survives a crash, the resurrected server (or the system administrator) must clean up any discrepancies between the allocation database and the status of the access files. The server should also confirm that its predecessor's clients are still active. This is significant programming effort to support an implementation that is still intrinsically vulnerable to faults.

4.2 *Add-on Fault Tolerance*

The fault-tolerant designs commonly described in the literature[4] are usually what I would call *add-on fault-tolerant.* They preserve many of the features of fault-intolerant implementation and they achieve fault tolerance by adding backups for components that are likely to fail. A revised allocator design might use a backup server and a duplicated allocation database. All transactions processed by the primary server are forwarded to the backup to ensure that the copies of the database remain consistent. When the primary server fails, the backup server assumes the role of primary server and arranges for another backup server to be started on a system that is still running. Despite the added complexity, this fault-tolerant allocator shares many of the problems with its fault-intolerant relative:

- *The server is still a bottleneck.* The revised design could add further delay while the primary server negotiates with the backup server to ensure database consistency.
- *Fault recovery is complicated.* The location of the primary server is now a variable rather than a constant: when a primary server crashes, the name of the new server must be sent to the network service manager(s). After a backup server assumes the role of primary server, the new backup server must create a new backup allocation database and the servers must synchronize their states. If there is a major system failure, such as a power failure in the machine room, the servers must be able to assess the damage and restart gracefully.
- *Network failures can impede operation and compromise fault recovery.* Remote procedure calls can still fail; complicated retry procedures might be necessary. If a network failure breaks the connection between the primary and backup servers both servers might try to recover independently. For full network partition this could be appropriate behavior, but a partial failure should not cause two servers to assume the primary role.

4.3 Intrinsic Fault Tolerance

When designing fault tolerant software one can avoid many problems by using a simple strategy: omit components that are likely to fail. Build a system that avoids faults instead of confronting them. The principle is already well-established in UNIX System software design: most programs process files as streams of bytes instead of trying to deal with multiple (and possibly incompatible) file formats. A resource allocator that must administer distributed resources also has an intrinsic potential for redundancy: systems may share some knowledge of inter-system allocations. This redundancy can add fault tolerance if properly exploited. Using these two guidelines, one might make the following design decisions:

- *A central allocation server may fail, or it may be a bottleneck.* Use a local server on each system.
- *Network links between systems may be unreliable.* Avoid unnecessary dependence on network communications. Clients should only make network connections to systems that own desired resources. Servers should execute commands locally instead of using remote command execution.
- *If a central allocation database is corrupted, all allocations on a network may be invalidated.* Avoid a single point of failure by distributing the database among the servers on the network.
- *Replicated allocation databases may become inconsistent.* Make no attempt to maintain duplicate copies of the database, have each server remember allocations that concern it directly. Servers can merge their versions of the database when necessary. Clients can serve as convenient messengers for sending allocation status to other servers.
- *Resynchronization after a crash can be complicated.* Don't try. After a crash, erase all memory of allocations on a system and depend on the other servers to remember the correct status. As long as one server is alive it can veto incorrect decisions made by its naive peers. Newly initialized servers will build a better approximation of the correct status when old allocations are released and resources are reassigned to new clients.

5. A Working Model

The remainder of this paper describes an implementation of a resource allocator designed according to the above guidelines. By external appearances, the new allocator resembles the simple allocator described previously: two commands, *alloc* and *dealloc*, allow clients to request resources and return them to the allocation pool. A typical *alloc* command is

alloc [*resource*] [*resource*] ...

where each *resource* is an optional argument that specifies a resource identified by file name (**-f** *file-name*) or by generic resource type (**-t** *type-name*). One can also use *system-name:resource-name* to indicate that a resource will be accessed on another system. For example: to ask for a magnetic tape drive on 'this' system, a tape drive on system *a* and tape drive *0* on system *b* one could enter

alloc -t mt -t a:mt -f b:/dev/rmt/0m

Deallocation is similar; either specify a list of assigned resources

dealloc -f a:/dev/rmt/1hn -t b:mt

or ask for deallocation of all assigned resources

dealloc -a

The superficial resemblance to the simple allocator is misleading, however. The following features avoid the flaws inherent in using an inappropriately simple model:

- *Deadlocks can be avoided.* Allocation requests are atomic; the *alloc* command waits until all requested resources are available before assigning any of them to a client. Clients who ask for all required resources in a single request are assured that they will not hold resources that are needed to satisfy other requests. Clients who insist on incremental allocation can still encounter deadlocks, but they have the option of avoiding the problem. The simple allocator, on the other hand, returns immediately after assigning only the resources that are currently free. This design is inherently prone to deadlocks that cannot be easily avoided.
- *Requests can be queued.* This is desirable if atomic requests are to be convenient. Without queuing, a request must fail completely if some resources are unavailable.
- *Multi-ported resources are protected.* The new allocator uses network software to communicate with allocation servers that run on all systems that own resources. The communication protocols ensure that all the owners of a resource know it is busy.
- *Remote allocation is possible.* This capability is part of the support for allocating multi-ported resources.
- *The allocator survives many system failures.* An intrinsically fault-tolerant implementation allows the allocator to operate reliably in the presence of most system crashes and some network failures.

6. Architecture and Operation

The allocator is implemented as a pair of programs: an allocation server and a client agent. These programs assume complementary roles: the server is the guardian of a system's nonsharable resources, the client agent negotiates with servers in behalf of clients who wish to use those resources. One instance of the allocation server runs on each system that supports resource allocation. Its functions are:

- Keep records of the status of the system's allocatable resources.
- Accept requests to allocate and deallocate resources.
- Accept status information from client agents, reconcile inconsistencies and update local status lists.
- Pass server status information to client agents.
- Change access permissions and ownership codes of resource files on the system.
- Monitor the activity of clients who own resources allocated by the system.

Multiple instances of the client agent may be active on a system, one for each client who is in the process of allocating or deallocating a resource. An agent has these responsibilities:

- Make network connections to allocation servers on all systems that have resources suitable for allocation or deallocation.
- Send allocation or deallocation requests to all connected servers.
- Accept status information from the allocation servers.

- Reconcile inconsistencies in the status information.
- Pass revised status information back to all connected servers.

Note the similarity of the client agent and allocation server functions. Each program gathers local information from several sources, combines it to construct a more global model and sends the revised information back to its sources. This exchange and consolidation of local information allows groups of servers and client agents to build global models and thus make reliable global decisions. Let us now examine some server and client agent activities in more detail.

6.1 Server Initialization

In principle there could be two server initialization procedures. All resources can be assumed to be free at system start-up, so a simple initialization would be appropriate. After a server failure some resources might be in use, so a more complex procedure could be necessary. The allocation server makes the simplifying assumption that these two cases are identical; all local copies of allocation status are discarded after a failure. In the rare cases where a server dies and the host system continues to function, allocation requests originated on that system are cancelled. Requests originated on other systems are unaffected; the other servers have duplicate copies of the lost information. Having avoided the complex problem of getting consistent status from other active servers, most initialization is trivial. A server must:

- Build a list of configured resources.
- Build a list of free resources by copying the configured resource list.
- Clear the list of active requests.
- Clear the list of granted requests.
- Advertize the availability of resource allocation as a network service.

The configured resource list is built after reading a configuration file. There are separate copies of the configuration file on each system, not necessarily duplicates; the file on a given system lists only the resources directly or indirectly accessible by that system. Each resource is described by an entry that contains the following items:

- *A list of generic names for the resource.* By convention, the names encode the capabilities of the resource. A magnetic tape drive that supports densities of 1600 BPI and 6250 BPI, for example, would have three names: *mt*, *mt1600* and *mt6250*. A client would use the first name to ask for a tape drive that supports any standard density; the other two names would be used if a specific density were required.
- *The negotiation group* – a list of the names of systems that have direct paths to the resource. The list identifies the systems that must be involved in negotiating any allocation of the resource. While this information could be derived from other items in the resource description, the list simplifies implementation and it serves as convenient documentation.
- *An 'on-allocation' action* – a block of text suitable for interpretation by the *shell*. The action is evaluated before assigning the resource to a client. One of the names of the resource will be in the **$RESOURCE** environment variable when the text is evaluated.
- *An 'on-deallocation' action* – another block of text suitable for *shell* interpretation. This action is evaluated after returning the resource to the free pool.
- *A list of resource names.* Each name has the form *system:file*, which indicates that *file* allows direct access to the resource on *system*. Access through a network file system such as System V RFS is indirect; the entry for a device file accessible by RFS would identify the host system.

6.2 Client Agent Initialization

Initialization of the client agent is context-specific; the agent is a program that implements the various options of the *alloc* and *dealloc* commands. When invoked as the 'allocate' option of *alloc*, the agent will:

- *Read the resource configuration file.* This is usually the same file that was read by the server, but it might be a newer version. The client agent and the server use entries common to the two versions to make allocation decisions.
- *Find the process id of the client's login shell.* After granting a request, the local server periodically confirms that a process with this id is still active. The id also serves to limit the scope of deallocation requests.
- *Gather other information that can be used to identify the request uniquely.* The servers use this information to avoid confusing similar requests.
- *Find all configured resources that are suitable for allocation.*
- *Merge the negotiation groups for those resources into a negotiation group for the request.*
- *Make network connections to allocation servers on all systems in the request negotiation group.* Connections are either *critical* or *optional*. If a resource will be accessed on a system, the connection to that system is critical; all other connections are optional. The agent aborts a request if it fails to make a critical connection.
- *Start the allocation protocol with all connected servers.*

Agent initialization when invoked as the 'deallocate' option of *dealloc* is similar:

- *Read the resource configuration file.*
- *Find the process id of the client's login shell.*
- *Connect to the local allocation server and ask for copies of its active and granted request lists.*
- *Find the most recent granted request originated by a child of the client's login shell.* This enforces the concept that all allocated resources are owned by a *login session* and avoids unintended interactions between concurrent sessions. A **-s** *session-id* option allows deallocation of resources allocated by other sessions.
- *Connect to all servers in the negotiation group for that granted request.* Connections to servers other than the local one are optional. Periodic fault recovery procedures eventually correct status inconsistencies caused by failed connections.
- *Start the deallocation protocol with all connected servers.*

6.3 Allocation

The allocation protocol is a variation of the two phase commit protocol;[5] it is designed to be fast and simple for the common case where a request can be granted immediately. It is also designed to tolerate some inconsistencies in the information available to servers and client agents. This section describes some of the strategies used to meet these goals.

A client agent begins the allocation protocol by sending

REQUEST *request*

to all servers in its negotiation group. The *request* describes the attributes of desired resources; it also supplies the system name, user name, session pid, agent pid and negotiation group. The servers may grant the request immediately, they may queue the request until the necessary resources are available, or they may start a conflict resolution protocol to resolve differences of opinion among the servers.

6.3.1 Immediate Allocation

When the local copy of a server's status indicates that a request can be granted immediately, the server's reply to the client agent is

ALLOC *request active-request-list granted-request-list*

All servers in the agent's negotiation group usually agree to allocate. As the agent receives these messages, it scans its local copy of the configured resource list and deletes any resources mentioned in the granted request lists. When all servers have replied, the configured resources in the remaining list can be assumed to be free on all systems in the negotiation group. If the agent can successfully allocate from this list, the reply to each server is

ACCEPT *request allocated-resource-list*

Each server confirms that the selected resources are listed in its local copy of the configured resource list. If this test fails, the server cancels the allocation and waits for a convenient time to read the latest version of the configuration file. Such events are rare. The servers usually mark the request 'granted', send an acknowledgement to the client agent and wait for the agent to sign off. Before acknowledging the allocation, each server scans the list of allocated resources to find ones requiring local access. For each local resource, a server evaluates the *on-allocation* action for the resource, changes the ownership codes for the appropriate access files, and gives read/write permissions to the new owner.

6.3.2 Queued Allocation

When a server receives a **REQUEST** for a busy resource, the response to the client agent is

WAIT *request active-request-list*

As in the case of immediate allocation, all servers in the agent's negotiation group usually make the same decision. The agent must merge the different versions of the active request list and then send the revised **WAIT** message back to the servers. If the revised message gives a server no new information, the server notes that a client agent is waiting for a response and takes no immediate action. If the server's local version of the status changes after merging it with the status in the **WAIT** message, the server returns another **WAIT**. The exchange of **WAIT** messages continues until the server sees no change in its status. When a *dealloc* command eventually frees the resource, each server sends **WAIT** to all client agents that are waiting for a response. If the change in status makes some agent eligible to allocate, the servers will return **ALLOC** when that agent echos the **WAIT** message.

6.3.3 Allocation in the Presence of Conflicts

Servers can make conflicting decisions: some return **ALLOC**, others return **WAIT**. Conflicts fall into two classes: timing conflicts and allocation status conflicts. Timing conflicts may occur when two or more clients ask for the same resources at about the same time. If the desired resources are distributed across several systems, message delay time may cause the servers on each system to give 'first arrival' priority to different clients. The servers must negotiate a single ordering for the requests. Allocation status conflicts can occur whenever the negotiation group for a resource does not include the system that originates a request for the resource. Since the originating system may never have been involved in prior negotiations, its server may assume that the resource is available when the other servers know it is allocated. A client agent must combine versions of the allocation status to make a correct decision.

6.3.3.1 Resolution of Timing Conflicts

Servers and client agents resolve timing conflicts by merging active request lists. Whenever an agent receives either an **ALLOC** or a **WAIT**, it combines its current version of the active request list with the one in the received message. The merge procedure compares the positions of requests in the two lists and orders the requests by their average distance from the tops of the lists; an arbitrary 'less than' relation picks an order in case of a tie. Servers use the identical merge procedure to combine **WAIT** messages received from client agents. As messages pass back and forth between servers and agents the different orders converge to a single value. When multiple requests arrive within the message propagation time window, the requests may be ordered arbitrarily. If the time between requests is greater than the message propagation time, requests follow a 'first come, first served' order.

6.3.3.2 Resolution of Allocation Status Conflicts

Client agents coordinate the actions of servers to resolve allocation status conflicts. Servers cannot easily resolve these conflicts, they have no direct communication paths to other servers. They can only make client agents pause when resources are unavailable. Agents synchronize server decisions so they become a series of votes. For each voting cycle an agent sends identical messages to all servers in its negotiation group and then waits for the responses. A server can exercise a veto by deciding to pause; the agent enforces the veto by refusing to send further messages until all servers have either responded or died. This strategy ensures that the allocator will pause until all surviving servers agree to allocate.

Requiring a unanimous vote has one unfortunate side-effect: allocation priority can sometimes be 'unfair'. If a server uses incomplete information, it may decide to allocate when other servers vote to pause. Once the server has decided to let a client agent allocate, it will not let another agent allocate until the first agent returns **WAIT** or **ACCEPT**. This may block allocation for new clients that are asking for unrelated resources; they must wait even though they should be allowed to allocate.

6.4 Deallocation

Compared to the complexity of allocation, deallocation is simple. Servers in the negotiation group for a deallocation get one message:

DEALLOC *request*

The *request* identifies the corresponding allocation and gives the names of resources being deallocated. The servers update their copies of the active and granted request lists and then send **WAIT** to all client agents that are waiting for a change in server status. Each server also finds deallocated resources that were being accessed locally, gives the access files to the *root* account, clears access permissions, and evaluates the *on-deallocation* actions.

There is one complication: sending **DEALLOC** to the deallocation negotiation group is not sufficient to ensure that all servers waiting for the deallocation are aware of the status change. Consider the case where resource X is connected to systems A and B, and resource Y is connected to systems B and C. If one client owns resource X, and another client wants resources X and Y, the server on system C will not get a **DEALLOC** message when resource X is deallocated; system C is not in the deallocation negotiation group for resource X. The solution requires a client agent to determine the *spanning negotiation group* for a deallocation. This is the union of the deallocation negotiation group and the active request negotiation groups for all requests on all systems in the deallocation negotiation group. The client agent must connect to all servers that are on systems in the spanning negotiation group but not in the deallocation negotiation group. Each of those servers gets a **WAKE** message, which tells the servers to send **WAIT** messages to all waiting client agents. For the above example, systems A and B are in the deallocation negotiation group for resource X. Systems A, B and C are in the spanning negotiation group since there is an active request for resources X and Y. System C will get a **WAKE** message because it isn't also in the deallocation negotiation group for resource X.

6.5 Strategies for Fault Recovery

The design of the allocator often lets it ignore faults instead of recovering from them. A system failure has no effect if the system's resources aren't needed to satisfy an allocation request. A network failure may be unnoticed if a client requests a local single-ported resource. When faults are inevitable, the allocator limits its actions to ones that ensure *partially correct* service: an allocation must either succeed, or fail completely. Some faults can be ignored: a failure to make an optional connection to a server is harmless, another server will have a redundant copy of the allocation status. Some faults can be corrected: a second attempt to make a critical server connection will sometimes succeed; servers occasionally send **WAIT** messages to idle clients to ensure that a fault in the allocation protocol doesn't cause an infinite pause. Other faults are fatal, the allocator aborts the transaction.

System crashes or server failures may erase parts of the allocation database, including records of successful allocations. Servers periodically start independent 'monitors', which look for signs of failures and try to repair the damage. Each monitor attempts to identify and kill allocations owned by dead shells, dead servers or dead systems (systems are only considered dead after repeated attempts to communicate fail over a fifteen minute interval). A monitor may also kill parts of allocations to reclaim resources made inaccessible by system crashes.

Servers check their environment periodically to ensure that they are making decisions based on current information. If a server's code or configuration list is out of date, it waits until it has no clients and then starts a new server.

7. Applications

The new resource allocator is intended to be a generic tool. It has been used to avoid security problems with tape drives, but it should adapt easily to other applications. The allocator may also serve as a nucleus for implementing other useful services. These examples should suggest possible directions to explore.

7.1 Software License Enforcement

Recent trends in software marketing complicate the life of a system administrator. Vendors sell products that are subject to restrictions in the number of configured systems or concurrent users. A license may also forbid using network connections to bypass the limits. The resource allocator could be used to simplify the enforcement of license restrictions.

A software license may be seen as an instance of an abstract resource. A license for a software package can be allocated if you create a file (perhaps a clone of */dev/null*) to serve as a surrogate for the license. The software would enforce its license by confirming that one of its license files is owned by the user of the package. For example; a license for *xyz* valid on systems *a*, *b* and *c* could be represented by a resource with the names *a:/dev/license/xyz*, *b:/dev/license/xyz* and *c:/dev/license/xyz*.

In practice, one would probably not wish to allocate licenses by file name; all licenses for a product should be equivalent, so no license should be preferable to another. Instead, a client would ask for a license by its generic name. The generic allocation feature also gives the license administrator flexibility when confronted with limitations in the terms of a license. Suppose, for example, that a licensed software product should be available on three systems: *a*, *b* and *c*. The terms of the license state that at most three clients may use the product at the same time, and only two clients may use it on a single system. To enforce these restrictions one could give this product a generic name of *xyz* and then create license files according to this table:

License	License File Configuration		
Number	System *a*	System *b*	System *c*
1	a:/dev/license/xyz.1	b:/dev/license/xyz.1	-
2	a:/dev/license/xyz.2	-	c:/dev/license/xyz.2
3	-	b:/dev/license/xyz.3	c:/dev/license/xyz.3

If necessary, the software could enforce a limit of one active program per client by using an operating system file lock facility; a second attempt to use the access file would fail. There is, however, nothing in the current implementation of the allocator that would limit a client to allocating only one license of a given type. Consider this a design simplification. The licensed software could usually enforce such a limitation by refusing to serve a client who owns more than one license. One might also use the *on-allocation* action associated with a resource to start a timer and use the *on-deallocation* action to stop the timer so clients could be charged for the time that they own each license.

7.2 Global Resources

Some peripherals no longer fit the model of being directly connected to a small number of processors. A laser printer with a network interface has equivalent connections to all systems on a network; it is a global resource. A software license would also fit this model if it permitted access by clients on all systems on a network. Representing this full connectivity in the allocator's resource configuration files would be inappropriate; a client agent would be required to make network connections to all allocation servers on the network. Fortunately, a full description is also unnecessary, an administrator can designate some small number of systems as the primary 'owners' of a resource and configure other systems as secondary owners. This is represented by having different versions of the resource configuration file on each system on the network. On any system that is not a primary owner, the resource is owned by 'this' system (a secondary owner) and by the primary owners. The entry for 'this' system lets a client make a request for the resource without knowing the names of the primary owners.

The choice of primary owners would be made after considering the network configuration and the desired level of fault tolerance. For the case where diskless work stations connect to a network file server, the server is an obvious choice; if the server is dead its clients are probably also in ill health. If multiple servers are available, one would select two or three to reduce the likelihood of losing all useful records of an allocation.

Keeping separate copies of the resource configuration file does complicate administration of the allocator. It would probably be desirable to maintain a central configuration database and then use software to compile system-specific copies and distribute them to each system. This would not add a central point of failure. The central copy would not be accessed directly, it would only be changed occasionally, it would usually be recoverable from backup media, and it could be regenerated from the distributed copies if necessary.

7.3 Operator Services

The original design proposal for the allocator included a companion 'mount this tape' facility. Because of a combination of user apathy and operator antipathy, implementation of the *tmount* command has been deferred. The command might be useful, however, for installations that limit public access to computer equipment. The allocator would require only minor changes to support a tape mount command. If the mount interface only needed to recognize absolute device file names the current allocator would suffice; one could enter the name manually. Generic allocation and mounting would probably require a mechanism for representing the concept of a 'partially allocated' resource;[2] on allocation, drives would be assigned but unavailable. When given a generic mount request the *tmount* command would select an allocated drive with the appropriate capabilities. After a successful mount the drive would be accessible.

An insecure implementation might be sufficient: the allocator could make removable-media resources inaccessible by clearing all access permissions. The *tmount* command could then ask the allocation server for a list of allocated resources and select one that has no access permissions set.

8. Problems

The resource allocator has usually performed as well as intended, but some human engineering and environmental problems remain:

- *Atomic allocation of multiple resources seems to be unpopular.* I have not correlated this with willingness to wear seat belts and shoulder harnesses, but I think I see a trend. A more likely explanation is that planning ahead is often inconvenient. Some users may believe a multi-resource

2. This is not a new concept. I first encountered it while hacking the TOPS-10 operating system sold by Digital Equipment Corporation. We added atomic requests and partial allocation to avoid frequent tape mount deadlocks among tape users.

allocation is a dispensable optimization rather than a desirable feature, others may not have read the documentation. As a concession to those who insist on asking for one resource at a time, the allocator will honor sequential requests that can be granted without waiting. There is no attempt, however, to make this convenient; allocations are stacked, not merged, and the most recent allocation is easier to deallocate than earlier ones. This feature avoids nasty implementation complexity at the cost of some frustration by confused users.

- *The queuing feature is rarely used.* Most of the explanation is simple: we have enough tape drives and we aren't allocating software licenses. Occasional comments that ''it just kept waiting, so I killed it'' suggest that waiting for programs is an unnatural act for UNIX System users who would rather negotiate with real people. The most recent version of the allocator only waits when explicitly given a **-w** option.
- *People frequently forget to deallocate resources.* This is no problem when someone turns off the terminal and goes home; a server will reclaim a resource when it notices that the login shell has died and the resource is not being accessed. Resources assigned to inactive sessions are another matter; social pressure does not seem to be sufficient incentive for timely deallocation. The allocator currently allows a minimum of one hour's inactivity before it reclaims an idle resource.
- *The concept of a login session is ill-defined.* Finding the process id of the appropriate login shell can be difficult when a client is using a multi-window terminal or a work station. The allocator makes a guess by reading the output of the *ps* command. It searches through the user's process tree to find the oldest ancestor still owned by that user. While this approach usually works, the processor time consumed by the *ps* command is usually more than the time used by the allocator. The search program must also be rewritten for each system that has a different *ps* output format. System calls that return the user id and parent pid of arbitrary processes would simplify this procedure; a supported formal definition of a session id would help even more.[6]
- *The UNIX System kernel may provide incomplete support for device control.* Some older versions of the kernel have no mechanism for forcing a tape off-line after deallocating the drive. This is a security breach. Even when the appropriate control operations are available, it is usually impossible to issue them if the device file is already 'open'.
- *Device and file status are often inaccessible.* It is difficult to ask the UNIX System kernel whether a file is in use, so the allocator must sometimes make arbitrary decisions when reclaiming resources. If a client allocates a tape drive, starts a tape-reading program in the background and then logs off, the allocator may assume the allocation is dead and reclaim the drive. The background process, however, can continue to use the drive until 'close'; a new owner will be unable to access it. The allocator avoids most of these incidents by protecting devices that have been accessed within the last ten minutes. It could make more intelligent decisions if it could know that a device is 'open'.
- *Network partition can cause incorrect behavior.* When two systems that own a dual-ported peripheral are unable to communicate, they will both allocate their ports independently. This is a side-effect of an attempt to be user-friendly. Since system crashes seem to be much more frequent than network failures, the allocator assumes that the system on the other port is dead if there is a failure to communicate with it. If 'correct' (but inconvenient) behavior were required it would be simple to add an option to cause the allocator to abort on any communication failure.
- *Client authentication is incomplete.* The allocator assumes that login names are identical on all systems on a network. This assumption is often true for a cluster of machines that are under common administration; it can fail when independent clusters are connected by a wide-area network. These conflicts can usually be avoided by careful administration of a system's resource configuration file; it should not contain entries for resources located where names would be ambiguous.
- *Inconsiderate users can subvert the queuing mechanism.* Since a resource file is owned by the assigned user, the group code and group permissions can be changed. Several users who are in the same group can agree to share a resource, excluding other users who are waiting. A System V user can also give a file away, which avoids the need to change group permissions. The problem could be eliminated by

giving each user a unique group code and giving allocated resources to *root*. The allocator could enable group access without granting the right of delegation.

9. Conclusions

As a tool, the allocator has been successful. Most people have used it with minimal documentation. Complaints have been rare. Some early confusion was eliminated by improving the error messages and making more intelligent decisions when reclaiming idle resources. Allocator performance is excellent; a request uses one to two seconds of processor time on our DEC VAX 8650 computers. The execution time spent finding the login session pid with a *ps* command often predominates. Real-time response is acceptable; a request is usually granted within four seconds.

During early design discussions some participants felt that separating allocation from tape mounting was inconvenient. Our experiences during a year of use justify the separation. Tape users have expressed no interest in a tape mount command, and operator reactions to any proposals have been negative. A tape mount facility at another local computer center is unused. If significant effort had been expended building a specialized tool, the work might have been wasted. Separating the functions has produced a general tool without unwanted accessories.

The attempt to design for intrinsic fault tolerance also seems to have been successful. Detailed traces of tests of the allocation protocol show correct resolution of timing conflicts that would rarely be encountered in practice. The incident logs written by the servers show no protocol failures, occasional communication failures caused by system crashes or malfunction of the 4.3BSD name daemon, and frequent recovery of dead or inactive allocations. I have no records of server failures caused by events other than system crashes, and the recovery from crashes has been uneventful and automatic.

I conclude with a summary of the strategies used while designing the resource allocator. They should be equally applicable to the design of other distributed services.

- *Keep it simple.* During initialization, a server avoids the difficulty of synchronizing itself with other servers by eliminating the concept. Software that doesn't exist is much less likely to be infested with bugs.
- *Exploit redundancy.* For transactions that complete without error, a central server with a redundant backup provides no better service than a single central server; possibly worse. The backup only buys reliability. Independent servers can *use* redundancy to give better service to local clients. Reliability is a desirable by-product.
- *Avoid unnecessary dependence on consistency.* The allocation protocol is designed to find consistent subsets of the information distributed among independent servers; inconsistencies are usually ignored.
- *Avoid unnecessary dependence on unreliable services.* Each allocation server assumes responsibility for any local actions required during allocation or deallocation. This eliminates the need to make new network connections or use remote procedure calls after granting a request. Error recovery is simpler; at worst, the server may have to retry a *fork* when attempting to execute a local command.
- *Use tools to avoid software faults.* The allocator was written in C++.[7] Function prototypes and strong typing exposed most coding inconsistencies before they could mature into adult bugs. The built-in data structure management in C++ eliminated most opportunities to mis-manage data structures manually. Source code compilation was controlled by the *nmake* software maintenance system.[8] The automatic maintenance of header file dependencies ensured that the executable files were in phase with the source code.

10. Acknowledgments

Jonathan Shopiro and Jerry Schwarz were invaluable sources of C++ advice, and Bjarne Stroustrup patiently supplied new versions of the C++ compiler when I broke it. Glenn Fowler suggested better ways to use *nmake*; he also added some features to make my work easier. Steve Bellovin contributed technical

suggestions and advice.

REFERENCES

1. Anonymous, "tape – user tape allocation interface," *UTS User Reference Manual*, 1986.

2. Olson, S.M. et al., "Concurrent Access Licensing," *Summer USENIX Technical Conference Proceedings*, pp. 287-294, June 1988.

3. Butterfield, D. (dave@lcc.uucp) and Woods, S. (scw@cepu.uucp), "get – a program to reserve exclusive use of a device," *net.sources*, 1985.

4. Cristian, Flaviu, "Issues in the Design of Highly Available Computing Systems," *Annual Symposium of the Canadian Information Processing Society*, Edmonton, Alberta, 1987.

5. Gray, James, "Notes on Data Base Operating Systems," IBM Computer Science Research Report RJ2188(30001)2/23/78.

6. Bellovin, S.M., "The Session Tty Manager," *Summer USENIX Technical Conference Proceedings*, pp. 339-354, June 1988.

7. Stroustrup, Bjarne, "The C++ Programming Language," Addison-Wesley, 1986.

8. Fowler, G.S., "The Fourth Generation Make," *Summer USENIX Technical Conference Proceedings*, pp. 159-174, June 1985.

A Unified Programming Interface for Character-Based and Graphical Window Systems

Marc J. Rochkind

Advanced Programming Institute Ltd.
Boulder, CO
(303) 443-4223

Abstract

Most UNIX programmers today sit at character-based terminals, and most of the user interfaces they develop, even for brand-new applications, are character-based. Yet, almost everyone agrees that future user interfaces will be graphical.

Unfortunately, few character-based applications will migrate easily to graphical window systems (such as X) because the graphical systems use event-driven programming models that are quite different from traditional character-based models, and because graphical systems use an entirely different application program interface.

The solution is to use a character-based window system that follows the programming model used by the graphical systems and that has the same programming interface. The Extensible Virtual Toolkit (XVT) interface is a good choice, because it already permits portability to several popular graphical window systems.

This paper reviews the evolution of UNIX user interfaces and explains why most developers can't use graphical interfaces today. Then it surveys the XVT programming interface and indicates the degree to which it can be mapped into a character-based model. Finally, the construction of the character-based system is briefly described.

EVOLUTION OF UNIX USER INTERFACES

UNIX was first developed, around 1970, for the most widely-used terminals of the time: 15 and 30 character-per-second hard-copy terminals. We think of UNIX as an interactive system, but most UNIX commands (e.g., **make**, **grep**, **ls**) aren't themselves interactive, although they are used during an interactive session. A few commands, notably **ed** and **sh**, do interact during their execution, but only in units of complete lines. This style is commonly referred to as *line-oriented* or *TTY* interaction.

By the mid-1970s CRT terminals with addressable cursors had largely replaced hard-copy terminals. But these were treated mostly as "glass TTYs." The traditional UNIX commands were still used, but more quietly and without consuming paper. An irritation was that their output often scrolled away before it could be read; commands such as **more** and **pg** were invented to deal with this problem.

No distributed UNIX commands from AT&T made use of cursor addressing. The first widely-used commands that did came from Berkeley, but still there were surprisingly few of them. The best known was of course the **vi** editor (later distributed by AT&T).

For the most part the BSD commands from Berkeley took over the entire screen; we'll refer to this style as *screen-oriented.* These commands didn't run in windows, although occasionally one (such as **talk**) split the screen.

Two subroutine packages were extracted from the screen-oriented software coming out of Berkeley: Termcap, which accessed a database of terminal capabilities, and Curses, which built on Termcap to sharply reduce the amount of redundant output to the display. Today Curses is the most popular way to interface to CRT devices, although the AT&T extensions to it that are shipped with System V have been plagued with bugs.

Because so few screen-oriented commands coming out of AT&T and Berkeley (the two main UNIX developers) take advantage of CRTs, one would not be far off even in 1989 to characterize the standard UNIX commands as mainly line-oriented. Some have taken to calling this old-fashioned interface "UNIX style," but as UNIX interfaces continue to improve that term will become increasingly inaccurate.

Not all UNIX software comes from AT&T and Berkeley, of course. There's a tremendous market for commercial packages to be used by end users who have no particular interest in UNIX for its own sake. Many of them aren't even aware that they are using UNIX. Ever since the early 1970s these applications have been screen-oriented. In the early days (beginning with "operations support systems" at Bell Labs [Lud78]) UNIX applications competed with those on other systems that used IBM 3270 terminals. Users expected a fill-in-the-blank form interface, not a line-oriented one. Later, in the 1980s, influences from PCs forced developers to go even further, incorporating windows, drop-down menus, and pop-up dialog boxes.

All of this line- and screen-oriented software is character based. The latest development is to equip UNIX computers with fast, high-resolution graphics screens and mice. Applications for these systems normally don't take over the entire screen, but instead run in windows. Developers don't deal directly with the hardware, as they often do with character displays, but instead develop for standardized window systems, such as the X Window System, NeWS, or one of the many proprietary systems such as SunView. We'll use the term *graphical-user-interface* for this style of interaction.

Interestingly, few technically advanced UNIX users make use of any graphical features of window systems other than the windows themselves. If one walks around a development shop with many workstations, one mostly sees huge screens with a **csh** window or two, a **vi** or **emacs** window, and maybe a **dbx** window. (Often the only use of graphics is for the ubiquitous performance meter!) That may be all programmers want to do with their workstations, but end users are clamoring for "Macintosh-like" applications with graphics, proportional fonts, scroll bars, menus, dialog boxes, and so on. And so, many of the programmers using the line- and screen-oriented tools from a decade ago are indeed developing state-of-the-art graphical applications. But, regrettably, seldom for their own use.

Where does all this leave us today? Programmers design mainly for one of three interface technologies:

- Programming tools and simple utilities are built to have line-oriented interfaces, because there's very little or no interaction, because Termcap and Curses hassles can be sidestepped, and because the resulting command can be used in a pipeline or with its input or output redirected.
- Commercial programs for widespread distribution, especially on inexpensive time-sharing systems running on Intel 80386-based PCs, use screen-oriented interfaces, mainly because

customers demand no less, and because costs prohibit more. (More reasons are given below.)

- Highly graphical applications, such as for CAD or publishing, require the capabilities of graphical user interfaces. Much, if not most, new development is targeted for the X Window System.

CHARACTER-BASED VS. GRAPHICAL USER-INTERFACE PROGRAMMING

Because today's commercial applications are strongly divided between character-based (but screen-oriented) and graphical user interfaces, it's worth looking at the technical, as opposed to marketing, differences between these two approaches. There are real differences due to hardware limitations and effective differences due to programming conventions.

Hardware Differences

The hardware differences between character-based and graphical user-interfaces are few but fundamental:

- Character terminals for UNIX rarely have a mouse (I have never seen one that did, although I have heard that they exist). This means that the entire interface must be operable from the keyboard.
- The coordinate system is in terms of character cells. Nearly all full-size terminals can display 24 rows and 80 columns, although 25 rows and 132 columns are also common.
- There is a fixed-width, unchangeable font. A few style variations, such as inverse video or underlining, are usually possible. Some terminals have a clever set of line-drawing characters that allow good-looking boxes to be drawn.
- Character terminals are usually quite slow, running at 1200 to 19200 bits-per-second. Sometimes, especially for small business systems, all the terminals are hard wired and therefore the designers can assume that they're fast. In other situations, however, the interface must be usable at slower speeds, to accommodate dial-up users. But even the faster speeds hamper designers who try to make applications look like those written for the IBM PC and its clones, whose displays run at an effective speed of about 110,000 bits-per-second.

Programming-Model Differences

Programming under a graphical window system isn't just a matter of calling graphics routines instead of **printf**. The whole programming model has changed from what most programmers—even those who have developed interactive systems—have become used to. This new approach has these key elements:

- *Event-driven organization.* Input to a program is more than just characters from the keyboard. There is a mouse, too. There are commands issued from a menu. There are requests to update a window that's been damaged by an overlapping window (possibly from a different application). There are inter-process messages. It's too awkward to poll for each of these inputs or to block on any group of them (they are not necessarily associated with file descriptors, so the **select** system call can't be used). A single queue of events neatly handles the situation. The main program becomes a big **switch** statement with a **case** for each event of interest.
- *Pixel-based, device-dependent coordinate system.* Most graphical window systems have a coordinate system based on device pixels (rather than a logical coordinate system). Characters and other graphical objects aren't located on a pixel (which is much too small), but rather are defined by boundaries made of mathematical shapes. The coordinate system isn't even isotropic—many displays have pixels that are higher than they are wide.
- *Graphics output primitives.* Output to a window resembles that provided by classical graphical packages. Pixel coordinates are specified, and possibly a pen and an interior fill pattern. Even text is drawn, rather than printed, and the application has to be concerned with the baseline and the leading as it goes from line to line.

- *Wide selection of fonts.* Instead of a single, monospaced font, there's a font library with fonts of different designs, weights, sizes, and styles. Generally, every symbol in every font has its own width, so even simple tasks like wrapping lines to fit between margins are complicated.
- *High-level commands.* In addition to standard ASCII characters, applications usually have to handle function keys (defined in terms of a virtual keyboard) and menu commands.
- *Dialog management.* Many user interactions occur in dialog boxes that contain controls such as push buttons, radio buttons, check boxes, scroll bars, typing fields, list boxes, and so on. To save programming time, these can be handled by a generalized dialog manager driven by a control list that contains specifications for each control. A callback function in the application program is called whenever a control is operated. Dialog managers do simplify the task of developing applications, but a program structured with appropriate callbacks is very different from its character-based equivalent, whatever that might be.
- *Resources.* Strings, menu definitions, user preferences, and the like are kept in resource files rather than being hard wired in the source code of the program. This makes it much easier to translate the application to other natural languages or to make minor interface changes.
- *Inter-application connection.* With a window system it's usually possible for the user to run several applications at once and to cut and paste information between them. Applications must specifically support these operations, following strict programming standards, if they are to work well with other tasks.

The impact of this new programming model on developers is that the job of moving applications from a character-based world is much tougher than would be required if only the hardware differences had to be taken into account. The situation could hardly be worse: Not only have most of the user-interface coding details changed, but the entire *structure* of the application is different. In most cases the application can't be modified to run on a graphical window system—it has to be entirely redesigned.

It's true, of course, that if the original application designers were farsighted enough to separate the user interface from the computational engine, then only the user interface module would have to be redone. But often it's not until designers experience the pain of moving to a window system that they recognize the benefits of this separation. What usually happens is that the character-based application is run as is under the window system until it can be reprogrammed. Because it's a lot harder to program for window systems than for character-based terminals, this reprogramming takes a long time.

WHY NOT JUST USE X?

Since the character-based and graphical programming models are so different, since practically everyone expects user interfaces to go graphical, and since X is essentially a universal standard, why not just start programming for X? That would prevent the horrendous conversion problems faced by developers who continue to produce character-based applications. And it would give developers the advantages of the superior programming model present in the graphical systems.

Well, there are a lot of good reasons why commercial developers don't want to use X:

- Hardly any UNIX hardware vendors have an X product, although nearly all have announced one. Usually a developer can get a preliminary version, but without support. (Sun users, for example, are expected to get their X system from the MIT release tape.) No responsible development manager would consider basing a serious application on such a precarious foundation.
- Although some developers are running X (mainly for experimental purposes), few end users are. Software vendors don't want to limit their market to customers who run X.
- X requires graphical displays, a mouse, and lots of computational horsepower (especially if the server and client are on the same machine, which is usually the case). Not everyone who wants to use X has the right equipment to do so. In most UNIX development shops pro-

grammers sit at 24-by-80 character terminals—only a few lucky ones have graphics workstations.

- Because of the special hardware requirements, the cost per station—which commercial developers are extremely sensitive to—is two or three times as much for X as it is for character-based technology.
- On hardware of the same power, X applications run slower than their character-based counterparts. Actually, only the user interface runs slower, but, because that's the part the user sees, the entire application is perceived as running slower.
- It's much, much harder to program for X than for character-based displays. Toolkits help a lot, of course, but X toolkits such as DECWindows, OSF/Motif, and Open Look Xt+ were released only this year. Character-based toolkits (of which there are many) have been available for years.
- Applications written for X aren't portable to PCs running MS-DOS or OS/2, or to Macintoshes.

MIGRATING FROM CHARACTERS TO GRAPHICS

Now, all of this sounds pretty bleak. Yet, every one of the problems listed will be erased with the passage of time. Maybe vendors aren't yet supporting X, but they will be very soon. The cost per station is higher for X, but several inexpensive X-server terminals will soon be available. In time, programmers and their managers will be comfortable with X, and it will no longer be considered too futuristic for them to bank on. All of the problems listed above can be summarized succinctly: *X is new*. That's certainly no crime—*all* systems start out being new!

So, what strategy should developers follow? Here are the possibilities:

1. Develop for X and suffer low sales until customers are ready for it.

2. Develop for character displays and ignore X, hoping it will go away.

3. Develop for character displays and redesign for X later.

4. Develop with an abstract, high-level toolkit that shields the character-oriented aspects of the user interface from the application, and then substitute an X-based implementation of that toolkit later, without changing the application.

Obviously, choice 4 is the one that makes the most sense. It's important that the toolkit be one appropriate for graphical window systems, not just for character-based displays, because otherwise the applications won't look properly graphical when they run on a graphical system. The point isn't to run the character-based application on a graphical system (you can already do that, in an **xterm** window). The point is to develop it using the graphical model (described above) but run it on character-based hardware. Remember, it's the model that's hardest to change. If it isn't right, the program can't migrate from characters to graphics.

We have developed the Extensible Virtual Toolkit (XVT), which is a portable interface to various graphical window systems; implementations of its uniform application program interface exist for the Macintosh, Microsoft Windows, and OS/2 Presentation Manager. An implementation for X is nearing completion [Roc89]. XVT is a good candidate for migrating character-based applications because:

- XVT is a much simpler interface than any window system offers. It restricts itself to what application programmers really need to do, rather than including lots of features to allow arbitrary customization.
- Because it has been used by developers for over a year, XVT has been proven to be a viable basis for serious commercial applications.
- XVT follows the graphics programming model (events, dialogs, etc.) as described above.

- Applications written for XVT can run on several window systems, not only X. It solves an even broader problem than the one proposed: Applications can move from character-based displays not only to X, but to other window systems as well.

MAPPING XVT TO CHARACTER-BASED HARDWARE

Can one really implement XVT on character-based devices? At first glance it seems questionable. Even though it's simpler than the popular graphical window systems on which it runs, XVT is still quite rich, and includes a range of features that in some ways exceed even those of X. For example, XVT includes mechanisms for WYSIWYG printing, while X does not.

To answer this question we went through each of the two-hundred or so XVT functions, placing each into one of three categories: easily implementable (e.g., **draw_text**), implementable with some work (e.g., **new_window**), and unimplementable or not needed (e.g., **draw_oval**).

Our most serious concern was the "unimplementable" category. If too many functions fell into it, especially important functions, then we really haven't successfully found a way to map XVT onto character-based displays. We might also conclude that, inasmuch as XVT is representative of graphical window systems in general (since it has implementations for many of them), too many unimplementable functions implies that there is *no* viable mapping from the character-based world of the present to the graphical world of the future. And that would mean that developers of commercial applications have no economic development strategy that would preserve their capital investment in their programs.

Happily, we found very few unimplementable functions, and most of them were in the area of inter-application communication. These are unimplementable only because we don't plan initially to offer a desktop environment that would manage multiple applications. Instead, each application will manage its own windows, and there will be no common system to cut and paste. (We may attack this area in the future.)

Even a few unimplemented function rules out 100% portability from any of the graphical implementations of XVT to the character-based one. Therefore we don't claim that applications can be ported in this direction, nor do we think many of our users are in a position to go this way (recall that the lack of a graphical interface was one of their problems—they want to move towards it, not away from it). We are concerned therefore only with portability from characters to graphics, to mirror the evolution we envision over the next few years of the technology and the user base.

In the next few paragraphs I'll review the main features of XVT and indicate how successfully we mapped them into a character-based implementation.

Events—XVT has 15 events for keyboard input, mouse actions, menu commands, scroll-bar operation, and window management such as activation, resizing, updating, and closing. All of them have character-based equivalents, although a few, such as resizing, aren't enabled in our initial implementation (since only the application, not the user, can resize a window). This doesn't impact portability—the application can still include a **case** for the event in its main event loop if it wants to.

Windows—Our design for character-based windows almost fully implements the XVT interface. They can be created, moved, resized, hidden, retitled, scrolled, and updated. The coordinate system local to a window is pixel-based in XVT, so we elected to use an 8-by-8 character matrix. For example, for a character in row 3, column 7 in a character-based coordinate system, we would say that it's at row 16, column 48 in the XVT pixel-based system. The application doesn't just assume that characters fit into a 8-by-8 bounding box, or that the display is 192 pixels by 640 pixels. Instead, it makes the same environmental queries that graphical XVT applications have to make to determine the display and font characteristics, and uses whatever numbers are reported. After all, a 8-by-8 character box is perfectly legal (albeit strange) on graphical systems, too.

Cursors and Carets—XVT supports a mouse cursor with changeable shapes, and a blinking caret to show the text insertion point. These are supported on character-based displays with a few modifications: The mouse cursor isn't changeable (so it can't be used to indicate a mode to the user), and the caret appears on top of or under a character rather than in-between characters.

Drawing—We decided that there would be little point in faking a graphics system. For example, we saw no need to emulate polylines with a sequence of vertical and horizontal lines. We couldn't envision any situation in which the result would be attractive (or even intelligible). So we implemented only those graphical operations that character-based displays can do well. There were only two: Drawing of text (in only one font and size, of course) and drawing of rectangles. Pens, brushes, and drawing modes aren't supported. While it may sound like little is available, in actuality enough is there to do anything that character-based displays can do, and with a set of functions that are much easier to use than Curses.

Pictures—XVT has the ability to capture a sequence of graphical and text output operations into an encapsulated picture that can be saved on a file, passed to other applications, and redrawn later as a graphic primitive. As the character-based implementation supports only text and boxes, and as no cut and paste was planned, we decided to skip support for pictures.

Clipboard—As mentioned above, there's no inter-application communication, and therefore no clipboard support.

Dialogs—XVT's dialog support is pretty lavish, with various kinds of buttons, list boxes, scroll bars, editable text, and labels. All of these control types are implemented in the character-based implementation. However, the control list that drives the dialog manager must be built up at run time. In some of the graphical implementations (such as the Mac, MS-Windows, and DECWindows) there is a resource manager that can read the control list from a resource file. We may implement resource files later.

Menus—XVT supports only a two-level menu hierarchy now. The top level is implemented as a menu bar (always visible) in most implementations; the second level of menu commands drops down from the bar. The character-based menus work exactly the same way, and all of XVT's menu interface is operational.

Printing—XVT prints by allowing the application programmer to open a special printing window and to then perform normal graphics operations on it. The character-based system works the same way, bearing in mind that there are only two graphics operations: drawing text and drawing rectangles. There's a simple driver that handles most character-oriented printers (including laser printers that have a monospaced font). We didn't have the resources to support anywhere near as many printers as the PC word-processor vendors do, so we made our driver extensible so we or our customers could add more support later.

Files—Because XVT is a portability tool, it provides some interface functions to remove file-access incompatibilities that exist across the various operating system and compiler combinations on which it runs. Fortunately, none of this is user-interface connected, so it all ported right over to the character-based implementation.

Memory Allocation—What was said in the preceding paragraph for files goes for memory allocation as well. The implementations for DOS and UNIX that we already had also worked in the respective environments for the character-based implementation.

Help System—XVT has a rather fancy online help system built in, designed to be embedded in applications. This is implemented portably in terms of XVT, and is already 100% portable between XVT versions. However, this doesn't mean that it would work on the character-based XVT, because this is an example of going from graphics to characters, which we said earlier wouldn't necessarily be possible. But we were lucky: The help system used only dialog boxes, which were fully implemented in the character-based system. So we didn't have to do anything to the help system except add some code to build up the necessary control lists at run time (recall that there is currently no resource file).

CONSTRUCTING THE CHARACTER-BASED XVT IMPLEMENTATION

We didn't have to build the entire character-based implementation from scratch. We already had a high-performance window system [Roc88] that has been used by hundreds of programmers on both DOS and practically every version of UNIX. This system has a user interface toolkit that includes a menu manager and some support for dialog boxes, so it seemed that much of it could be used, perhaps with only a new interface.

But, alas, we were in the same situation as anyone else who attempts to go from character-based systems to graphical window systems: We weren't using the right model. Among other things, our existing window system wasn't event driven, and the dialog manager was inflexible (it didn't use a control list). So merely constructing a new interface, treating the existing system as a black box, was out of the question.

The window system in [Roc88] consists of three layers. The *physical screen* layer handles the entire screen, and provides an interface to higher layers with a very limited, but entirely sufficient, set of primitives: You can output text at any location with any attribute, you can fill a rectangular area with any character and attribute, you can scroll a rectangular area in any of four directions, and you can position the cursor. (One can think of the physical screen layer as implementing a RISC terminal.) There are five implementations of the physical screen interface: for the Z-19 terminal, for Termcap, for Curses, for the IBM PC BIOS, and for the IBM PC memory-mapped display (users have developed more, for systems such as VMS and OS/2).

Next comes the *window* layer, which implements overlapping rectangular windows, each with a local character-oriented coordinate system. The fundamental operations at this level supplied to higher levels mirrors those of the physical screen layer: text output, filling, scrolling, and cursor positioning. When an obscured part of a window is uncovered it calls an application-specified callback function to perform an update of the damaged area.

Finally, there is the *virtual screen* layer, which implements rectangular areas, possibly larger even than the screen, that lie behind a window. Applications never get an update event (there's backing store, to use X's terminology), and they can freely position the cursor anywhere—the system scrolls the virtual screen automatically.

For the character-based version of XVT we used the physical screen and window layers intact, with only cosmetic changes, and tossed out the virtual screen layer, because its special (and attractive) features just weren't part of the XVT model (maybe they should be, but that's another matter).

The software in [Roc88] also includes an extremely general table-driven keyboard interface that allows any function key on any terminal's keyboard to be mapped into a defined virtual key code. It does this by building a finite-state machine at application-initialization time, and then running each character through the machine until a final state is reached (function keys usually issue several characters). We found this ideal for generating XVT keyboard events, which also use virtual key codes, and used it without change.

Only one other component of our existing user interface toolkit could be used in XVT: the menu manager. It already implemented a menu bar (at the top of the screen, not in the windows) and drop-down menus. We changed it around a bit (it used virtual screens, which we were about to eliminate), but were able to salvage most of it, including all of the really tricky stuff.

All other parts of the character-based XVT implementation had to be programmed fresh. This was lots of work, but also pleasurable work. For the first time since we've been developing XVT implementations we didn't have to fight some vendor's window system—all of the code from the operating system on up was our own! For this reason the character-based version turned out to be the most efficient, most reliable, best understood, most maintainable, and most easily supported of all the versions we've done to date. So, who said character-based systems were bad?

CONCLUSIONS

We've successfully created a character-based window system and user interface toolkit that's competitive in size and speed to other contemporary character-based tools, but that uses the event-driven, dialog-box oriented programming model common to today's graphical window systems. It meets the XVT interface, which has already been demonstrated to be compatible with many of the commercially important window systems.

The result is a smooth upgrade path for software developers who need to be character-based now, but want to move to the emerging graphical systems later. They can build a better user interface than they usually do (because few character-based window systems are as complete as XVT), and they can become graphical whenever they want without reprogramming. As a bonus, programmers can begin training themselves for the event-based programming style of the future without getting bogged down in the complexities of programming for X, the Macintosh, or Windows/PM.

We've also demonstrate a more general result: Although there's no denying the *hardware* differences between character-based and graphical technology, the *programming-model* differences are gratuitous. Graphical window systems don't deserve a monopoly on event-driven application design, and users stuck with character-based terminal don't deserve inferior user interfaces.

REFERENCES

[Lud78] Luderer, G. W. R. *et al.* "The UNIX Operating System as a Base for Applications," *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, July/August, 1978.

[Roc87] Rochkind, Marc J. *Technical Overview of the Extensible Virtual Toolkit (XVT)*. Boulder, CO: Advanced Programming Institute Ltd.

[Roc88] Rochkind, Marc J. *Advanced C Programming for Displays.* Englewood Cliffs, NJ: Prentice-Hall, 1988. ISBN 0-13-010240-7

[Roc89] Rochkind, Marc J. "XVT: A Virtual Toolkit for Portability Between Window Systems," *Proceedings of the Winter 1989 USENIX Technical Conference.*

At Home With X11/NeWS Windows

Mark Opperman
opcode@sun.com

Sun Microsystems, Inc.
Mountain View, California

ABSTRACT

The X11/NeWS server has been modified to support X11 and NeWS clients across a serial interface providing remote access to window-based applications. The UNIX system services required by the X11/NeWS server have been isolated into a *helper* process that runs on the host to which the window server connects. A reliable, full-duplex, channel-based protocol provides the link between the window server and the helper process. Pre-allocated channels are used for system services (e.g., *open*, *close*, *spawn*) and dynamically allocated channels are used to multiplex data streams from the different window client applications. Data transmitted between the window server and the helper process are compressed using the adaptive Lempel-Ziv technique. Prototype versions of the server run on an ATARI ST and on Sun workstations with surprisingly good performance for both NeWS and X11 clients at baud rates as low as 2400.

Background[1]

In early 1986, long before the recent advent of X terminals, a group at Sun was looking at the possibility of building what they hoped would eventually become a standard graphics terminal for UNIX-based systems. At the same time, another group at Sun was completing their work on the first version of NeWS (the Network extensible Window System). The two ideas were merged together into an effort dubbed the NUT (variously attributed to either "Non UNIX Terminal" or "NeWS User Terminal"). The idea behind the NUT was to provide a low-cost, bitmapped terminal interface to UNIX based on NeWS. Communication with the UNIX host would be provided either via the ethernet or via a serial interface.

About the same time, ATARI introduced their first ST computer, the 520ST, and it seemed to provide the minimum requirements of a serial NUT—a bitmapped display, an 8 MHz 68000 CPU, a two-button mouse, and sufficient memory (when upgraded to 1 Mb). A port of NeWS to the 520ST was undertaken as a demonstration of the window system's portability. This port, which was shown at the introduction of NeWS in October 1986 at the Sun User Group (SUG) meeting, provided the foundation and inspiration for the current work.

Although some work continued on the NUT effort after the SUG meeting, the initial work finally was abandoned. Memory leaks in the NeWS server had posed severe constraints on non-VM systems, and, as is often prone to happen in a rapid growth industry, priorities changed.

However, in early 1988, with the increasing popularity of both X11 and NeWS, a renewed effort was undertaken to overcome some of the technical problems encountered in the earlier work:

- Reliable network access across a serial interface.
- A stand-alone, or ROMable, NeWS server.
- Login authentication.
- Serial line performance.

This work resulted in a new version for the ATARI MEGA ST4[2] (4 Mb RAM) based on NeWS 1.1. A reliable serial protocol was developed, a subset of fonts was built into the server in order to bootstrap the terminal, a NeWS-based login window was created, and font files were compressed across the serial interface. The system was starting to become useful at 9600 baud and would stay up for days at a time (assuming known memory intensive operations were avoided). Still, it was seen that there was room for further performance improvements.

The work described here is based on this further evolution of the NeWS 1.1 NUT prototype using a newly developed serial protocol, and is integrated with Sun's X11/NeWS server. We will look at the overall architecture of the system, some details of the serial protocol and what we term the *helper* process, changes to the X11/NeWS server for the NUT, and the design of NeWS clients for low-bandwidth terminals.

Overview

The focus of our work was to modify the X11/NeWS window system such that, instead of requiring high-bandwidth networking like ethernet, it might perform well over a relatively low-bandwidth communication link. We chose the RS232 interface as the physical layer because of its widespread use in the industry, especially in terminal applications, its potential for use in wide-area networking, and its compatibility with most PC class systems. Our ulterior motive was to develop a platform with which we could have exactly the same window environment at home (running remote applications across a modem) as we do at work.

X11 and NeWS

The X Window System and NeWS both address the problem of creating a single, common interface to network-based window applications that can run on a wide variety of hardware platforms. However, the approaches taken by X and NeWS differ radically—the X protocol is based on a static procedural interface resembling an RPC mechanism whereas the NeWS protocol is simply the PostScript language (with extensions to handle multiple canvases and input) and the server is an interpreter. Both protocols have their inherent advantages and Sun's new window platform will be based on a server that understands both protocols.

Client Communication

One of the characteristics that X and NeWS share is the need for a reliable, bi-directional byte stream between client window applications and the server. One common protocol used to provide this byte stream, and that used on Sun workstations, is TCP/IP. Another characteristic common to both window systems is the need to access files containing font information. PostScript generalizes file access to include the ability to read and write *any* file, and NeWS extends this functionality to special types of files, in particular those providing network access. Further, NeWS also provides some extensions that are rather UNIX-specific, e.g., *getenv*, *putenv*, and *forkunix*.

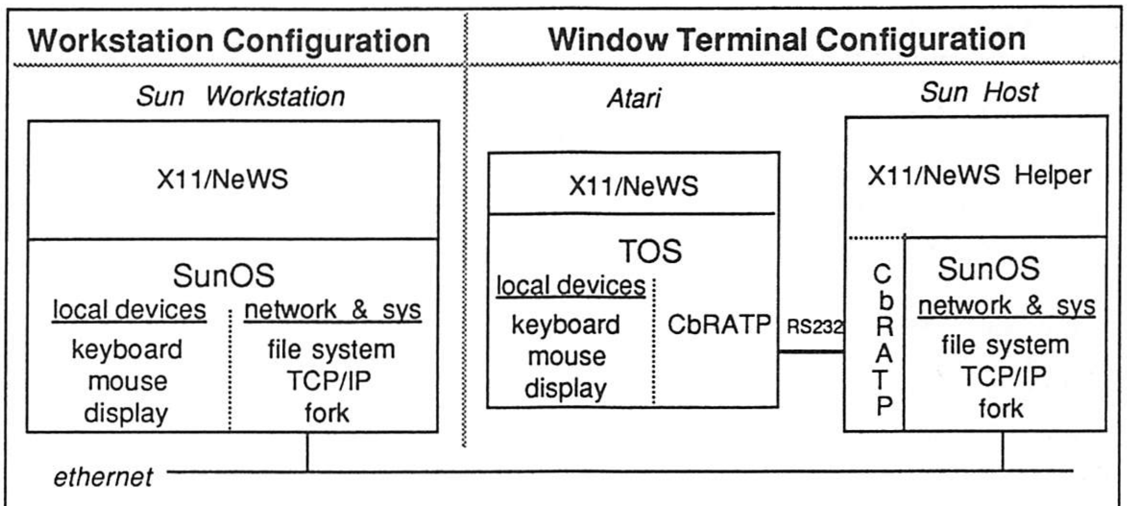


Figure 1. Simplified X11/NeWS Architectures

NUT Approach

The architecture of the serial-based X11/NeWS server grew out of these requirements—to provide multiple, reliable, bi-directional byte streams between the server and clients and to provide the system services needed by the window system. The approach taken was to develop a Channel-based Reliable Asynchronous Transfer Protocol[3] (CbRATP) where some channels are statically allocated to provide system services and other channels are dynamically allocated to clients as they make connections to the server. A *helper process* running on the host system actually implements the system services on behalf of the window server. The protocol is full duplex so that clients can be sending requests to the server at the same time the server is writing data or performing system services. The advantages of this architecture lie in the simplicity of the window terminal software (essentially no operating system is necessary) and in the efficiency of the serial protocol.

In a network with serial-based X11/NeWS servers, window client applications would communicate with the window system in exactly the same way as in an ethernet-based system. Instead of directly opening sockets to listen for clients, the window system sends requests to the helper process to open the sockets on its behalf. Upon initialization, ports 6000 and 2000, defaults for X11 and NeWS, respectively, are opened by the helper. To window clients, the helper process appears exactly as the standard X11/NeWS server. Once a connection is established, two channels are allocated for bi-directional client/server communication. Client processes can run on any host on the network that has access rights for the window system.

Helper Process

The helper process has two functions—to provide the system services required of the X11/NeWS server and to multiplex the data streams from the window clients across the serial interface. It uses the Channel-based Reliable Asynchronous Transfer Protocol (CbRATP) for both functions. The services provided by the helper process can be grouped into five principal functions: file services, network services, system services, protocol services, and compatibility services. These are shown in the Table 1.

The file services are used for standard file access: font files in the case of X11 and any user-accessible file in the case of NeWS. The requests correspond to the system calls provided by SunOS and most other disk-based operating systems. The FileSize request generates a *fstat* system call on SunOS, but only returns the file's size. Statically allocated channels for reading and writing are not required as these channels are allocated dynamically.

Network services are required to establish connections between window clients and the server. OpenConnection is used to listen for client connections. AcceptConnection is called when a client connection has been established. ServerAddress and ClientAddress are used to check access privileges. GetHostNames is used to implement the X Request *ChangeHosts*, and GetHostAddresses is used to implement the X Request *ListHosts*.

The system services are rather UNIX-specific. Getenv and Putenv are required by NeWS to establish search paths for fonts and executables, and to allow clients started from the server to inherit the information needed to connect back to the server. The GetDTableSize service is provided so that the server can be extended to support systems with local disk files. On UNIX, the Spawn service invokes a *vfork* system call.

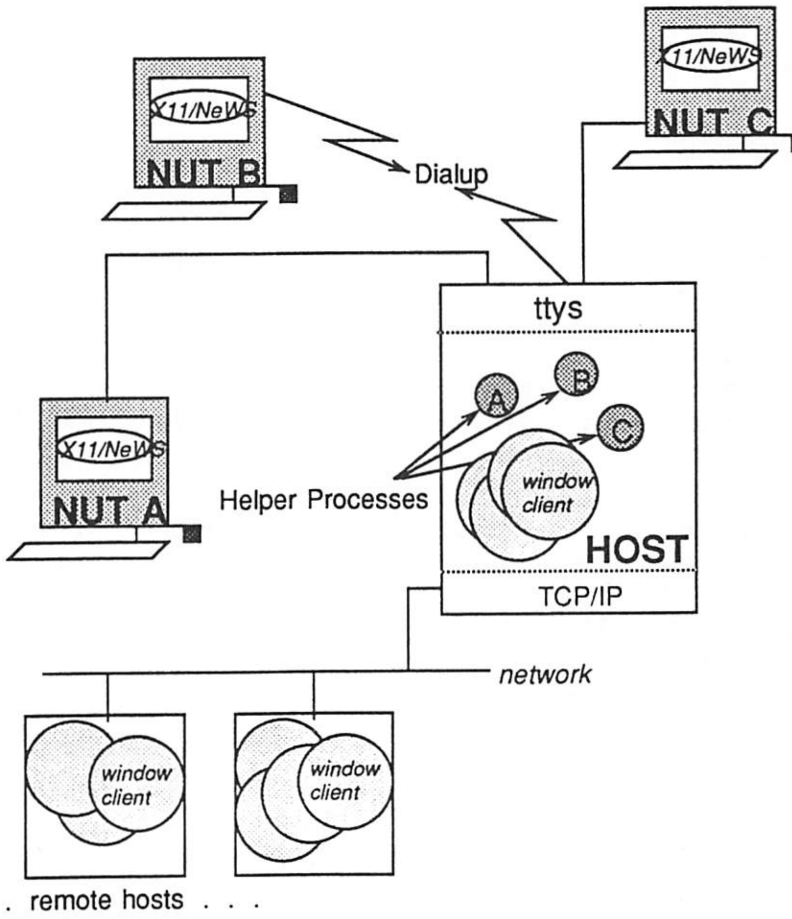


Figure 2. Serial X11/NeWS Window Servers

The protocol services provide support for changing the baud rate on direct connections, selecting the use of a Crc in the packet protocol, and Quiting the helper process. In normal operation the CRC is always used, but the request was added to measure the performance difference over a reliable link layer.

The compatibility services provide support for the NeWS operator *enumeratefontdicts* used in NeWS 1.1.

Rather than try to create a one-to-one correspondence between NeWS system calls and the services provided by the helper process, the initial helper was designed to provide a level of abstraction for those services required by a network-based window system. This design was confirmed by the fact that only two additional services were required to support the addition of the X11 protocol—GetHostNames and GetHostAddresses.

File Services	**Network Services**	**Protocol Services**
Open	OpenConnection	Quit
Close	AcceptConnection	Speed
Lseek	ServerAddress	Crc
FileSize	ClientAddress	
	ServerName	
System Services	GetHostNames	
Getenv	GetHostAddresses	
Putenv		
HomeDir	**NeWS 1.1 Compatibility Services**	
CurrentDirectory	InitFontSearch	
GetDTableSize	NextFont	
Spawn		

Table 1. Request Types

In our first implementation, we have developed a helper process that runs on SunOS 4.0 across the Sun workstation platforms. Work is currently underway to port the helper to a System VR3 environment. Because all of the operating system dependencies are now isolated in the helper process, we expect it to be quite straightforward to port the window server to other hardware platforms. Similarly, as long as an operating system can provide networked connections, file system access, and emulation of the system services, the helper can be ported to it.

Serial Communication Layer

The Channel-based Reliable Asynchronous Transfer Protocol (CbRATP) provides the basis for all communication between the window server and the helper process on the host. It is a packet based protocol providing pre-allocated channels for system services and dynamically allocated channels for reliable, full-duplex multiplexing of the client connections.

We would have preferred to have chosen a standard asynchronous protocol, but we did not find any that met our criteria for performance and availability. X.PC was rejected due to the limited number of channels available and its lack of acceptance in the UNIX marketplace. RATP[4] was rejected because it only allows one outstanding packet in each direction, uses a simple checksum, has slightly more overhead than we needed, and also is not widely available. SLIP was rejected for several reasons: high per-packet overhead of TCP/IP headers, no single standard version, no data compression, and no error detection at the packet level. The IETF point-to-point (PPP) protocol[5] appears to be an emerging standard that could provide the functionality we need with only slightly increased packet overhead. Because of this, we have chosen compatibility with the PPP protocol at the framing level.

CbRATP packets consist of a frame start/stop byte, a control byte, a channel byte, an arbitrary number of data bytes (currently limited to approximately 256 in practice), a two byte CCITT CRC-16 checksum, and a frame start/stop byte. For back-to-back packets, only one inter-packet start/stop byte is required. Start/stop bytes occurring within a packet are escaped by a special escape character. The protocol requires an eight bit data path.

The packet control byte includes a two bit SEQ that indicates the cyclic identifier of the packet and a two-bit ACK that indicates the cyclic identifier of the last packet received. Four control bits indicate a SYNC packet (S), a RETRY packet (R), a Lempel-Ziv compressed packet (L), and a Continued packet (C).

```
  7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
 +---+---+-+-+-+-+---------------+---------------+
 |seq|ack|R|S|L|C|    channel    |   ...data...  |
 +---+---+-+-+-+-+---------------+---------------+
 |             CRC-16            |  start/stop   |
 +---------------+-------------------------------+
```

Figure 3. CbRATP Packets

Packet Layer

The packet layer handles transmission failure and retransmission. An ACK is received for each packet sent, although up to four packets may be sent in either direction without acknowledgment. This enables several small packets to be sent while a large packet is being received. The protocol guarantees that the data transmitted at one end, when received, arrives unaltered and intact at the other end.

Associated with each packet is a cyclic identifier [0..3]. This ID is used to maintain order and to acknowledge received packets. When a packet is successfully received, its ID is returned in the ACK field of the next outgoing packet. If several packets are received before an ACK can be sent, only an ACK for the last packet received need be sent. An ACK implies that all packets up to the ACK have been received. The ACK is usually inserted in the ACK field of the next outgoing packet. If no packet is queued for transmission, then a special SYNC packet is sent. SYNC packets are also sent whenever the line has been idle for too long.

The packet layer can be viewed as two distinct sides—the reader and the writer. Both sides are processed in parallel for full-duplex line utilization.

Channels

The channel layer passes data between a user process and the packet layer. Each channel has a function associated with it that specifies how the data is interpreted. In the NUT, pre-allocated channels correspond to each of the system services provided by the helper (see Table 1, above). For example, when the NUT sends the string "NEWSHOME" on channel 13, the helper invokes the function associated with channel 13 (*getenv*), and returns the results also on channel 13. Dynamically allocated channels are assigned to client applications as they connect to the server and whenever a file is opened. The function associated with these channels continually sends data to the other side as long as there is data available for reading.

Compression

One key to the success of CbRATP in this application is the compression it provides for nearly all data transmission. Using the Lempel-Ziv algorithm[6] with 12-bit codes, the data transmitted is often compressed to just 25% of its original size. We chose not to use larger code sizes as our analysis showed that the marginal gain in compression was small compared to the additional memory requirements of the algorithm. Using 12-bit codes requires only 30 Kb of static data, whereas using between 13-bit and 16-bit codes would have required 54 Kb to 414 Kb bytes of static data. When looking at the compression of font bitmaps, there was often no improvement in compression for the more common small font sizes, and only minimal improvement for larger font sizes. Not all packets are compressed, only those with four or more bytes of data. Although up to 20% of the data packets may have less than four bytes of data, the amount of data in these packets represents less than 1% of the total data transmitted.

Overall, this protocol provides the requisite functionality with very little overhead. Because the framing level is identical to that for the proposed Internet PPP, we hope that a STREAMS interface to the packet layer will be forthcoming and that it will provide improved performance. The current bottleneck in the helper process is in the large number of system calls required to read each packet.

Compression Examples

We looked at the compression performance of a couple of the common X11 applications, *xclock* and *xterm*, and their NeWS counterparts, *roundclock* and *psterm*.

When xclock is started with a second hand it transmits 128 bytes to the server every second. After running for just a couple of minutes, data compression results in about 30 bytes being transmitted each second. By contrast, roundclock normally transmits 74 bytes each second and this is compressed to 10 or 11 bytes after running a few minutes. It's worth noting that these numbers depend on the data that have already been transmitted as this affects the state of the compression algorithm. Theoretically, if the same data are sent continuously, it will eventually compress down to a single code (9-12 bits).

When typing into an xterm, each keystroke generates a 32 byte *KeyPress* event from the server and a 60 byte *ImageText8* request from the client. The *KeyPress* event is typically compressed to between 7 and 14 bytes. The *ImageText8* request can be compressed to 16 or fewer bytes for single keystrokes. When type-ahead occurs, xterm sends character strings in the *ImageText8* request for better throughput. Nonetheless, when echoing a single character, the effective baud rate is reduced to 1/16th of the actual baud rate. However, when *cat*ing large files, xterm throughput is increased significantly (see the section on performance).

In the case of the standard X11/NeWS version of psterm, the server transmits one byte (the keystroke) to the client, and psterm transmits 39 bytes of PostScript back to the server to redraw a single character and move the cursor. The single keystroke to the client is not compressed, and the reply is typically compressed to 10 to 15 bytes. With the packet overhead, this produces an effective baud rate 1/13th of the actual baud rate.

Clearly, improved performance was required for interactive use of a terminal emulator. How we achieved that performance is described in the section on *Low Bandwidth NeWS Clients*.

Serial Window Server

The major changes made to the X11/NeWS server were in the following areas:

- CbRATP support.
- Login terminal window.

- PostScript startup files in server data segment.
- Subset of fonts in server data segment.

In addition, the Atari port included:

- A virtual screen that doubles the effective size of the Atari display.
- The device-dependent graphics library support.
- Keyboard, mouse, and serial line drivers.

We have also added support for multiple NeWS servers per host and have modified the NeWS terminal emulator, psterm, to reduce its client to server communication requirements.

Login Terminal

When the serial X11/NeWS server boots, a special login window is automatically displayed. The window is a simple terminal emulator, written entirely in NeWS, that permits direct communication through the serial port. The login terminal runs as soon as the first step of server initialization has completed. The user logs in using the normal login procedure and then invokes the helper process. When the helper process runs, it switches the serial line discipline to raw mode and initiates the CbRATP protocol on the line. Once this occurs, all data is transmitted in packet mode until the user exits the server. In packet mode, the server continues its initialization, retrieving the list of fonts available on disk, opening the sockets for client connections, and reading the user's startup file, etc. When this initialization has completed, the terminal is closed and the server is ready to accept connections for any X11 or NeWS client. One advantage to this method is that no network system administration is required for the X11/NeWS terminal—it appears as just another ASCII terminal to the system.

PostScript Startup Files

Normally, the X11/NeWS server reads and executes a PostScript file, *init.ps*, upon startup. Init.ps in turn loads over 20 other X11/NeWS initialization files. In the case of a serial NUT, this initialization data had to be included within the server data segment as we assume no local file storage. We accomplish this with slight modifications to the standard startup files that allow us to run init.ps through a preprocessor that replaces some of the standard startup with Nut-specific startup. This file is then run through *cps*[7], a standard NeWS utility, to reduce the size of the string by replacing system keywords, strings, and numbers with tokenized representations.

One disadvantage of placing the startup files in the server data segment is that they can no longer be modified. For most users, this is not a problem. However, two files are often modified by users—*user.ps* and *startup.ps*. User.ps is read at the very end of server initialization and this has not changed in the serial NUT. Startup.ps, however, is normally read very early in the initialization process. It cannot be read in the serial NUT because files cannot be read until after packet mode has been initiated, and this does not occur until after the helper has started. In a window terminal this could be addressed through the use of an EEPROM.

ROM Fonts

We have developed a mechanism whereby a subset of bitmapped fonts can be specified in a configuration file and then automatically built into the server. Until the helper process is invoked and disk-based fonts are accessible, all requested fonts are retrieved from the server's data segment. Once the helper is run, the available fonts are updated and references to font families are retrieved from disk. Whenever a font bitmap is requested that can be found in the data segment, that bitmap is used instead of reading it from disk.

Atari

One attractive feature of the Atari port is the use of a virtual screen that effectively doubles the screen size of the Atari monitor from 640x400 to 640x800. The Atari framebuffer is of a simple memory map design which utilizes a portion of the CPU main memory. It also has the feature that any part of that memory can be displayed by just changing a pointer to the new display memory location. The serial X11/NeWS server takes advantage of this by initializing a screen twice as high as the normal Atari display. When the cursor is moved off the top or bottom of the physical display the screen is smooth scrolled to follow the cursor until it reaches the top or bottom of the virtual display. This actions occurs at the mouse interrupt level and is independent of the server event mechanism.

The Atari server was actually compiled and loaded on a Sun-3 workstation. A tool was written to convert from the Sun executable format to Atari's TOS format. Our turn-around time was decreased significantly by the fact that the Atari can read DOS format floppies and we could write DOS floppies using PC-NFS. Although most of the system dependent functionality is located in the helper process, we still needed to write drivers for the Atari keyboard, display, mouse, and serial interface. Much of this work was done in assembly language.

Multiple Servers per Host

Prior to the X11/NeWS server, NeWS did not easily support multiple invocations of the server on the same host. It was possible, but inconvenient at best. Because the helper resembles a NeWS server, and there are potentially many of them running on the same host, it was necessary to add this mechanism to the server. Rather than write a network style daemon to register NeWS users and their host/port addresses, we took the simple approach used by X11 to cycle through available port addresses, beginning at a well known address. This functionality was added to the server with only a few lines of PostScript in init.ps (see below). In addition, we modified the program *setnewshost* to easily select the host/port combination.

```
% Returns the 'file' used for client connections.
% First try NeWS_socket, then NEWSSOCKET environment variable,
% then the reserved port #144, and finally the default port #2000.
% If all fail, then continue trying ports 2001, 2002, ... until
% successful.
%
/registerserver {      % - => file
   {
      { /NeWS_socket load } stopped
      {                % not found, try environment
         pop           % /NeWS_socket => -
         { (NEWSSOCKET) getenv } stopped
         { pop (%socketl144) } if
      } if
      (r) file
   } stopped
   {                   % still not found, cycle through ports
      pop pop          % filename (r) => -
      2000 1 9999 {    % start at default port 2000
         { (     ) cvs (%socketl) exch append (r) file } stopped
         { pop pop }
         { exit } ifelse
       for
   } if
} ?def
```

Low Bandwidth NeWS Clients

One motivating factor behind the development of a serial-based NeWS server was the belief that NeWS clients could take advantage of the extensible nature of the NeWS server, thereby reducing the amount of PostScript sent across the wire and improving interactive performance. Although we found this to be the case, we also discovered that many existing clients either do not take advantage of this feature or simply used the bandwidth poorly.

Psterm

Our first target for improved client performance was *psterm*, the NeWS terminal emulator. We found that in the simple case of echoing a single keystroke, psterm would transmit 39 bytes of data (many of which were redundant) to the server:

```
( ) x y CD 1 1 x y ER newpath x y moveto (c) show ( ) x+1 y CU
```

where the routine CD takes the cursor down, ER erases a rectangle, and CU does a cursor up. Although the system tokens (*newpath*, *moveto*, and *show*) were being tokenized by cps, the user-defined tokens CD, ER, and CU were being sent to the server in ASCII. The first step was to use the cps usertoken facility to reduce each of these strings to one byte, thus reducing the PostScript string to 33 bytes.

The next step was to determine how to eliminate the redundant information being sent to the server on each keystroke. The approach taken was to do some simple pattern matching on the psterm output buffer each time it was being flushed to the server. When the (now) 33 byte "echo character" sequence was spotted, we were able to reduce it to something that resembled:

```
(c) FE
```

where FE does a fast echo. By defining FE to be a usertoken, another two bytes were saved. This reduced the data sent from psterm to three bytes when echoing a keystroke. We were able to do this by keeping in psterm the current (x, y) cursor coordinates each time the CU routine was invoked. With packet overhead, this means that 14 bytes are sent to echo a single character. Although this reduces a 2400 baud connection to an effective 300 baud for interactive work, the CbRATP layer will coalesce outgoing data packets to reduce the packet overhead and increase the throughput. With these improvements, interactive performance of psterm at 2400 baud is acceptable.

General Considerations

Another common mistake of NeWS clients is to always download their own utility routines to the server regardless of whether or not the client has already run. Unless the routines are explicitly stored in `systemdict`, they end up in the client-specific `userdict`. `Userdict` is garbage collected when the client terminates. Although there is a memory trade-off, these routines can instead be stored in a new dictionary in systemdict. When a subsequent client starts up, it can check for the existence of this dictionary and avoid reloading the routines, greatly reducing the startup time.

Most clients use the *cps* utility to construct tokenized PostScript that is sent to the server. By default, *cps* will convert system tokens (e.g., *arc*, *moveto*, *stroke*, etc.), integer, fixed point and floating point numbers, and strings to more compact representations. For example, system tokens are represented in one or two bytes. However, cps will not tokenize user defined tokens unless explicitly directed to do so. There are 32 one-byte usertokens and 1024 two-byte usertokens available. Usertokens are especially effective during interactive operations where immediate feedback is required. As an example of what not to do, we have seen one client that sends (among other things),

```
PaintHiLight clearselectionpath
```

each time a key is pressed. By defining a PostScript routine that invokes the two routines and associating a usertoken with it, the above data can be reduced to a single one-byte usertoken.

Monitoring

In order to find the kind of problems described above, we developed a *curses* program that displays in real-time the communication between NeWS clients and the X11/NeWS server. The program opens a client connection to the server and then listens on another port for "real" NeWS clients. As data is transmitted in either direction, it is displayed by the monitor. This program is available on a SUG tape of contributed software and is called *newsmon*.

Serial X11/NeWS Performance

Memory Requirements

The static size of our prototype server (based on pre-β X11/NeWS source code) is 917 Kb text, 461 Kb data, and 91 Kb bss. The data segment includes a 270 Kb *init.ps* string and 109 Kb of bitmap fonts required for startup. The fonts include Courier (10 and 12 pt), Times-Roman (12 pt), Times-Italic (12 pt), Cursor (12 pt), fixed (13 pt), Helvetica (12 pt) and Helvetica-Bold (12 and 14 pt).

Although we expect to further reduce the size of the server, our prototype now requires 3.0 Mb after initialization and running one psterm. Although we have not yet adequately addressed the problem of the server running out of memory on a non-VM system, the X11/NeWS server group is currently looking at various ways to recover from out-of-memory errors, for example, by reclaiming memory from the font cache and by disabling retained canvases. For now, the server does require a 4 Mb memory configuration.

Startup Times

To give some idea of the performance we are seeing with our prototype system, below are the startup times for some typical applications at different baud rates These numbers were taken from a pre-β server running on a diskless Sun-3/50 directly connected to a helper process running on a Sun-3/110.

	9600 baud	2400 baud
Psterm	7-8 sec	9-10 sec
Xterm	11-12 sec	15-16 sec
Roundclock	2-3 sec	3-4 sec
Xclock	8-9 sec	13-14 sec

Table 2. Client Startup Times

As can be seen from the table, startup times for clients running at 2400 baud are not necessarily four times greater than when running at 9600 baud. There are a couple of explanations for this behavior: 1) there is a certain amount of fixed overhead per packet regardless of the baud rate (the protocol is implemented in user mode) and 2) a client's performance may be more compute-bound than bandwidth-bound, thus overshadowing any difference in serial line speed.

Cat /etc/termcap

The following table shows the elapsed time when executing *cat /etc/termcap* from the X11 and the NeWS terminal emulators, as well as from a Wyse-50 terminal. Note the surprising result that this is actually **faster** using the window server. This counter-intuitive result can be explained by the fact that the line speed is the bottleneck and with our compression we are actually transmitting fewer bytes across the line to the terminal emulators. The throughput from these tests at 9600 baud show 887, 1180, and 1292 chars/sec for the Wyse-50, psterm, and xterm, respectively.

	9600 baud	2400 baud
Psterm	1 min 55 sec	8 min 35 sec
Xterm	1 min 45 sec	9 min 15 sec
Wyse-50	2 min 33 sec	10 min 12 sec

Table 3. Cat /etc/termcap Times (135,752 bytes)

Future Directions

Our work has shown that the X11/NeWS server can be ported to a system with very little operating system support and with only a low-bandwidth network connection. This has been accomplished by off-loading the system services required by the server to a helper process on a remote UNIX host. We see this work as being important to applications of wide-area workstations, dedicated window server terminals, and window servers built on PC class machines. We also see the following areas as important for future implementations.

- Support for local file I/O on systems with local disks, e.g., for local font storage or NeWS clients.
- PostScript process pipes so that the server could be used stand-alone for PostScript development.
- A network-based name server for locating a user's window server.
- Support for multiple logical network access from the window server.
- STREAMS based CbRATP protocol module.
- Full Internet PPP protocol conformance.
- ISDN support.
- Tools for developing low-bandwidth applications.
- Full use of outline fonts.

Acknowledgments

Sun's X11/NeWS server group, led by Steve Evans, has done an outstanding job building the window system upon which this work is based. Chris Berry and Mike Dill have developed much of the NUT-specific software described here. Tom Maurano has kept the vision of a low-end window server, and hence the project, alive. David Rosenthal and Owen Densmore provided excellent suggestions for improvement in the paper. And finally, Mitch Bradley must be credited with originating the idea of a helper process.

References

1. Mitch Bradley, personal communication, February 1989.

2. Atari Corporation, *Atari Mega ST Computer Owner's Manual*, Sunnyvale, CA, 1987.

3. Chris Berry, *Channel-based Reliable Asynchronous Transfer Protocol*, Sun internal document, September 1988.

4. G. Finn, *Reliable Asynchronous Transfer Protocol (RATP)*, RFC 916, DARPA, 1984.

5. Drew Perkins, *The Point-to-Point Protocol (PPP): A Proposed Standard for the Transmission of IP Datagrams Over Point-to-Point Links [DRAFT]*, Internet Engineering Task Force, Carnegie Mellon University, December 1988.

6. Terry A. Welch, "A Technique for High-Performance Data Compression", *IEEE Computer*, pp. 8-19, June 1984.

7. Sun Microsystems, *NeWS Programmer's Guide*, Part Number 800-2379-04, Mountain View, CA, January 1989.

Incorporating Usability Studies and Interface Design into Software Development

Kate Ehrlich, Barbara Stanley[1] *and Tim Shea*

Sun Microsystems Inc.
Entry Systems Software
2 Federal St
Billerica MA 01821
(katee@East.Sun.COM)

Abstract

The new graphical file manager on the Sun*386i*[2], Sun Organizer[3], was designed to provide easy access to file system and network operations for people unfamiliar with the Unix[4] operating system. The graphical interface to Organizer supported a direct manipulation style of interaction, a clear graphic design and plenty of user feedback. In order to verify that Organizer was easy to use by the target audience, we conducted a study in which a representative group of users followed a self-administered tutorial. This introduced them to basic operations such as changing directories, selecting files, deleting and adding files and moving/copying files between directories. Although these users were enthusiastic about Organizer and completed the tutorial successfully, there were several instances where the behavior of the application did not match the users' expectations. Many of these mismatches were corrected in the final product resulting in a smoother, more efficient interface. Users also appreciated the emphasis on active learning. For novice users especially, being given an explicit set of procedures to follow can help overcome limits in the rate of assimilating new information.

1. Introduction

When systems were designed with the computer rather than the user in mind, applications were often hard to learn and hard to use. Building on advances in system performance, bit-mapped screens, window systems and input devices, interface designers have developed graphical interfaces that make computer systems easier to use [2, 13].

However, new systems can still be hard to learn. Carroll and Mazur [5] studied professionals as they learned to use the Apple Lisa system, which at the time of the study, was considered state-of-the-art in interface design and ease-of-use. The people they observed had numerous difficulties. These included problems with the documentation, inability to follow the on-line tutorials, difficulty learning the interface conventions, and difficulty using a mouse.

[1]Barbara Stanley is now at Petra Software, 53 Marblehead Rd, Windham NH 03087.

[2]Sun386i is a trademark of Sun Microsystems, Inc.

[3]Sun Organizer is a trademark of Sun Microsystems, Inc.

[4]Unix is a registered trademark of AT&T.

The point of this example is to underscore the benefits of usability testing, especially when the testing can be done early enough for the results to be incorporated into the product. A good interface design goes a long way toward making a system easy to use. But even the best design cannot anticipate all the ways a system is used. For example, in their book on Human Factors, Rubinstein & Hersh [10] describe a study of a system designed for children. To keep the interface simple, the children used a light pen to select objects on the screen. In the study it was found that several children would stick the light pens in their ears. The wax would block the photocells preventing the cells from working. While this problem is understandable, it was not anticipated and was only uncovered by testing.

This paper describes a usability study done on Sun Organizer, the new graphical file manager on the Sun*386i* workstation. The study was conducted after a major re-design of the interface but while the product was still under development. Feedback from the study led engineering to make further improvements in ease-of-use. The paper concludes with a discussion of the importance of making usability testing an early integral step in product development.

2. Software Design

Sun Organizer was targeted at application/commercial users (e.g. CAD designers, electrical engineers, architects, research scientists) who should not be required to learn the Unix operating system in order to be productive using the Sun*386i* workstation. Although these users are not computer scientists and do not necessarily know the Unix operating system, they tend to be technically sophisticated with high expectations of productivity. For example, non technical users might have a large number of files to maintain; they might want to share work files with peers; or they might want to move around the file system and explore the directory hierarchy or the network.

With these target users in mind, the technical goals of the project were:

- Provide an easy-to-learn interface to the Unix file system and network.
- Provide a complete set of functionality that would also be efficient to use.
- Provide a programmatic interface to administrative applications (e.g., file back-up and restore.)

File types were a crucial component of the design. With file types, users/system administrators can specify which application Opens, Edits, and Prints the files of a specified file type. For example, to Edit an icon file, users want an "iconedit" editor to be invoked -- not their default text editor. Because the Unix operating system does not support file types well, how Organizer should support file types was constantly debated. The two major proposals were: (1) the use of magic numbers, and (2) file type based on the file name. Because reading magic numbers is slow (i.e., the file must be opened) and the use of magic numbers was not yet widespread, the latter proposal was chosen. Thus, the user/system administrator may specify a regular expression to match the file name.

To simplify the network commands, a new system feature, the "automounter" [3], was developed. The "automounter" automatically mounts an NFS file system when a user tries to access a file or directory on it. Two automount points are always available: "/home" and "/net". "/home" provides transparent access to other users' home directories -- local or remote, while "/net" provides transparent access to directories on remote systems. When users Open "/home" or "/net", Sun Organizer reads the yellow pages, rather than simply reading the directory as it does in most cases. The result is a transparent network user interface that is consistent with the hierarchical structure of the Unix file system.

Organizer provides two methods for users to view the file system: Map Display mode and List Display mode:

- Map Display mode, shown in Figure 1, shows users a "map" of the filesystem. It allows them to see several directories at once. In the example shown in Figure 1, the user Opened the directory "memos" and then the directory "misc" to display their contents. The current directory, /home/mtravis, has not been changed.
- In List Display mode, the user can see the contents of only one directory at a time. However, in this mode the contents of the directory are displayed in a more efficient multi-column layout allowing the user to see more files in a single window than in Map Display

The design and implementation of Organizer drew on previous work [15, 16]. It was also designed to be compatible with the standard SunOS[5] desktop environment and SunView[6] applications. The result was a SunView window application with a scrollable graphic display area that contained the file names and icons, a button control panel at the top of the window for the most commonly used operations, and pop-up menus and property sheets as part of the interface. Although this interface had a number of ease of use features (e.g., multi-level undo) the graphic design was not clear. In particular, the icons were confusing and the information was poorly laid out both in the button panel and in the display. The next section describes the changes made to the interface design.

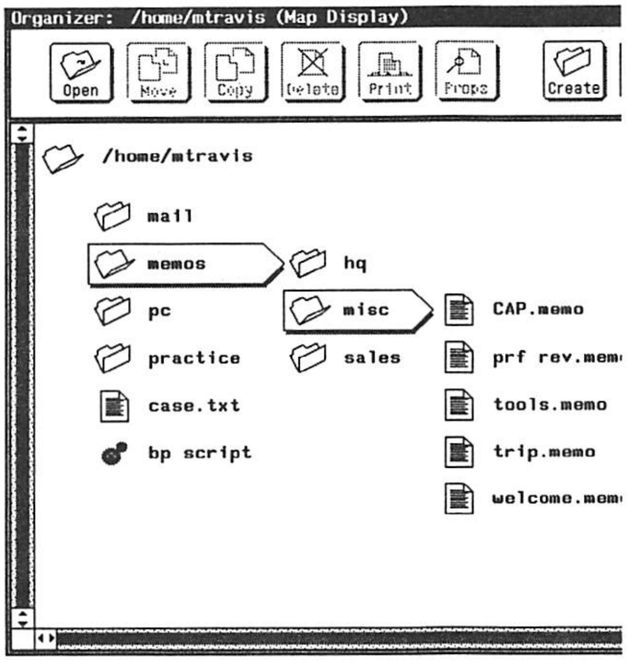


Figure 1: Snapshot of an Organizer window (Map Display mode)

[5]SunOS is a trademark of Sun Microsystems, Inc.

[6]SunView is a trademark of Sun Microsystems, Inc.

3. Interface Design

The improvements to the original design made use of well researched principles of interface design. These included: increasing the use of a direct manipulation style of interface [11], providing users with better feedback on their actions and on status information [14]; and, in general, clarifying the appearance and layout of the interface elements to simplify information retrieval [12]. Below we describe some of these changes.

Increase direct manipulation

In a direct manipulation style of interface the user performs some action directly on the object instead of going through the intermediary of a command language or even a menu selection. Examples of this style in Organizer include:

- Double-clicking on a directory in the pathname. To change to a higher directory (in List Display mode), the user double-clicks on that directory in the pathname.
- Rename by typing over the existing name. Users could rename any file or directory very simply by first selecting that file (single click on the name) and then typing in the new name. This would overwrite the original name.

Provide better feedback to users

In the re-design we placed considerable emphasis on providing users with feedback, either about the status of the application (e.g., whether they were in List Display or Map Display Mode), changes in state (e.g., opening a directory) or general feedback on the user's actions (e.g., highlighting files when selected). Examples included:

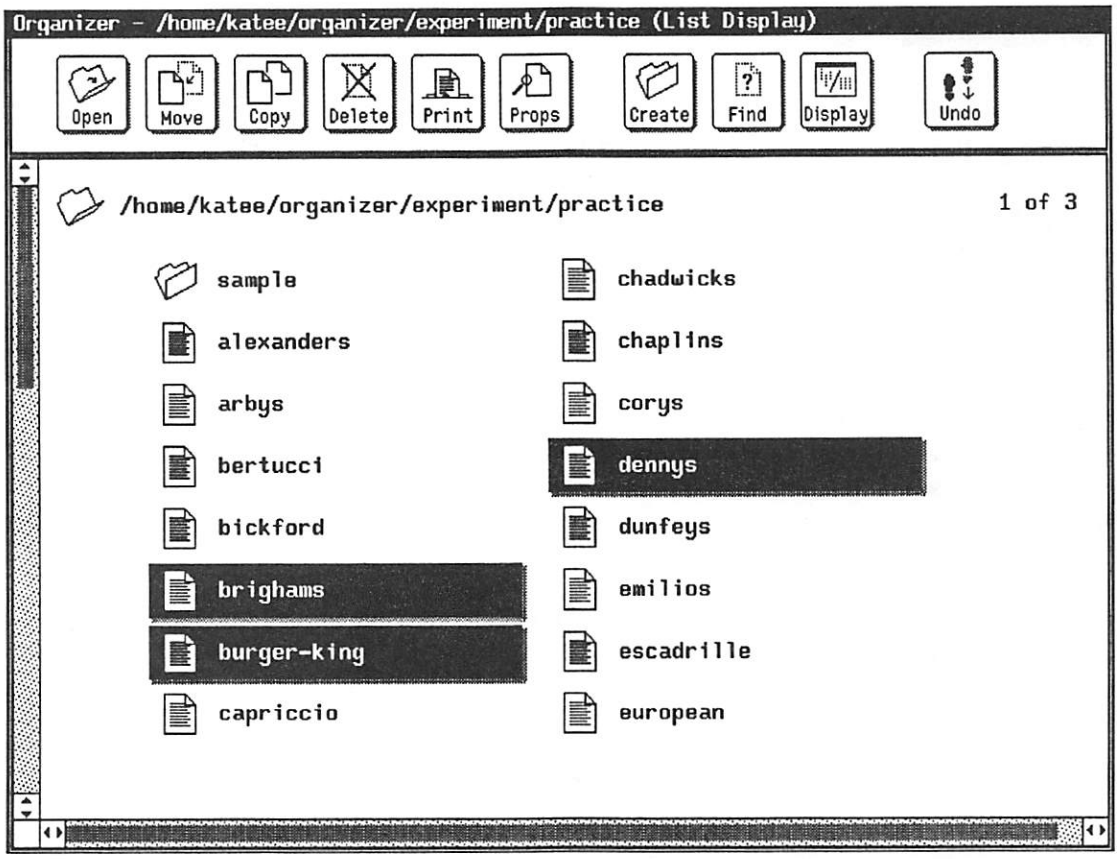
- The directory icon was animated when opened. When a user opened a directory, the icon would change from a closed directory to an open one. For instance, in Figure 1 compare the icon next to the directory "mail" and the one next to the current directory /home/mtravis, shown at the top of the listing; "mtravis" is open. Also, during a Move or Copy, the directory opens to "accept" the new file and closes again when the operation is complete.
- Filenames and icons were highlighted when selected. The graphic design of the file highlighting was changed to give the selected file more emphasis than in the previous design through the use of color and shadowing. Also, the graphic design of the icons was changed to emphasize the different file types.
- Buttons for file operations were grayed out when not available. When a user selected a file, all the buttons would be highlighted indicating availability. If no file was selected or the user did not have the right permissions, the set of buttons that applied to file operations (Move, Copy, Delete, Print. Props) would be grayed out indicating that they were not available.
- There were page numbers in List Display Mode. The upper right hand corner of the screen displayed the current page number (see Figure 2).
- The name stripe displayed status information. The name stripe at the top of the application window showed the name of the current directory and the current mode (List Display or Map Display).

Provide clear graphic design

An important but frequently overlooked part of interface design is making sure that the elements are grouped in a way that users can find what they need with very little searching. Mostly, this affects ordering of elements in a menu-based interface. However, even in our graphical interface, we tried to minimize the user's search in the following ways:

- Buttons were added to the display. To keep as little as possible hidden from the user all user actions could be initiated from one of the buttons at the top of the menu. This was a change from the previous design in which several commands were available only from hierarchical pop-up menus.
- Buttons were reorganized into sensible groupings. Undo was placed on its own at the far right of the button display. The fixed commands: Create, Find and Display were grouped together as were the commands which applied to selected items: Open, Move, Copy, Delete, Print, Props. These operations were also available on a pop-up menu, in the same order as they were displayed in the buttons

Many of these features are illustrated in Figure 2.



Organizer - /home/katee/organizer/experiment/practice (List Display)

/home/katee/organizer/experiment/practice 1 of 3

sample
alexanders
arbys
bertucci
bickford
brighams
burger-king
capriccio
chadwicks
chaplins
corys
dennys
dunfeys
emilios
escadrille
european

Figure 2: Example of an Organizer window (List Display mode) used in the study.

4. Usability Testing

Goals of the Study

Although the interface was carefully crafted, we wanted to confirm that the interface was easy to use by our target audience of non-technical end-users. The usability study was designed to:

- *Determine acceptance.* We considered the product to be acceptable if people could do useful work within a single training session (approximately 1 hour).
- *Identify "bugs".* These were instances where the behavior of the system was unexpected or inconsistent from the user's point of view. Most of the enhancements to Organizer were made as a result of finding such "bugs".
- *Identify problems and misconceptions.* This category reflected more serious problems users had understanding the "rules" of the interface and the data model.
- *Observe learning strategies.* The hour long session provided an opportunity to focus on problems such as those identified in the LisaLearning study, including use of the mouse, ability to comprehend the instructions and the effect of 'active learning'.

Design of the Study

In the study we observed people as they learned to use Sun Organizer on the Sun*386i* workstation. The 14 people who participated in the study were recruited primarily from people who work in the Sun facility in Billerica, MA. For the most part these people were familiar with using Sun workstations (and hence with the command line interface to the file system) and had heard of and probably seen Organizer (a few had even used it a few times). They volunteered for the study because they hoped to use Organizer when the development was completed. Thus, these Subjects were already motivated to learn the application

It was important that Subjects be given realistic and representative tasks to perform. Subjects in the study learned Organizer by following specially written instructions that described basic operations. For each of these operations, Subjects first read a brief paragraph and then performed a scripted sequence of actions that was an example of the description just read. Below is the script that accompanied the description for selecting multiple files; the result of the first three operations is illustrated in Figure 2.

- *Click left on "brighams".*
- *Press and hold down left as you move the mouse button through "burger-king".*
- *Click middle on "dennys".*
- *Click left on empty space in the window.*

Subjects could read and follow these instructions at their own pace and they were free to refer back to a topic at any time. Special directories were set up so that the instructions and the information on the screen matched. Most people completed this initial training phase in 20-30 minutes.

Upon completion of the training, Subjects were given two sets of simple problems to test how much they learned in the training phase. They then filled out a questionnaire on their opinion of Organizer and on their professional background. Subjects' behavior was recorded on paper with an audio tape for backup. The entire session lasted approximately 1 hour.

5. Results of study

Acceptance

The Subjects were very enthusiastic about Organizer. One of the more experienced said "People who have used Unix are not used to being treated so kindly." They especially liked the direct manipulation style of interface that let them navigate up the hierarchy by clicking on the directory name in the pathname and that let them rename a file by simply typing over the existing name.

Subjects left the session able to use Organizer on their own. By our criteria this meant that Organizer was "acceptable" which meant that it did not require major changes. However, based on some of the "bugs" we found, described below, there were several modifications that needed to be made to Organizer before it was ready to be released.

Bugs

"Bugs" were simple things that might go unnoticed or be dismissed as temporary problems. It was our impression, however, that these simple mismatches between the users expectations and the systems' behavior subtly undermined the user's confidence in the system and therefore should be corrected before the product was released.

- Users were surprised that the action of double-clicking to open a directory in Map display did not have a corresponding double-click to close. Instead users could close a directory by opening another one.
- Users wanted confirmation of actions especially after operations that Deleted or Moved files out of the current directory.
- If a user mistakenly tried to copy to a file instead of to a directory that file destination remained highlighted giving the user inconsistent feedback that the file selection was okay.
- The redraw was too slow which reduced users' confidence in the application.
- The scroll keys sometimes didn't work when the user was in the middle of a move or copy operation.
- Users were prevented from deleting a directory when it was not empty.
- The bounding box drawn after the user selected multiple files sometimes deselected the first file.
- One of the more interesting "bugs" we found was in Map Display. A user would sometimes inadvertently do a recursive copy of the destination's ancestor directory. This happened because the arrow around the selected directory extended too far into the column listing the contents of that directory. When a user selected a file at the top of the column to copy into the current directory (as they were asked to do in one of the exercises) they sometimes, unknown to themselves, also selected the parent directory marked with an arrow. Thus they ended up trying to copy that directory into itself. Fortunately in this case, the recursion timed out and the user just ended up copying more than intended.

Difficulties

Users did not understand the difference between List Display and Map Display mode. In List Display, double-clicking on a directory will change to that directory. In Map Display, double-clicking on a directory will display that directory's contents but does not change the status of the current directory. Most of the Subjects had considerable difficulty understanding this difference. In particular, several subjects did not know what was the current directory when asked. The difficulty is perhaps not surprising when one considers that there is both tactile and visual feedback consistent with the misconception.

- *Tactile.* In List Display, users double-click to change a directory. But in Map display the same double-click on a directory name will display its contents but not change the current directory. Most beginning users were not be sensitive to this subtle difference in result for the same action of double-clicking.
- *Visual.* In Map Display, when the user double-clicks on a directory, an arrow is drawn around that directory to indicate the current selection and the contents are listed in the next column, frequently beginning on the same horizontal level as the parent directory. Most users mistakenly took these visual cues as indication that the selected directory was also the current directory. However, the current directory remained, as it had been before the arrowed directory was selected, as the one listed in the ancestor list at the top of the page.

Note that there were several cues to the current display: the name stripe at the top of the window stated the display type; List Display showed page numbers in the right hand corner; List Display typically (but not always) displayed the contents of a directory in 2 or 3 columns whereas Map Display always listed the contents of a directory in a single column (see Figure 2). These cues, however, were clearly inadequate for conveying the different modes to users.

Learning

This section focuses on 4 components of the learning process: Mechanics of learning the interface, especially the mouse; following instructions; timing of information; and, the role of 'active learning' [6].

Mouse. Several people had some difficulty using the mouse, especially learning to double-click correctly. To change directories or to open an editor on a file in Organizer, users double-click (two clicks on the left mouse button in quick succession) on the selected file or directory. This was a new paradigm to many users and everyone mis-timed the double-click at least once (Carroll & Mazur observed similar problems with people learning the Lisa). The difficulty with double-clicking was only an annoyance but it was persistent. In a less structured learning environment, an inexperienced user could easily misinterpret the failed double-click, or worse, become frustrated with the application and give up. In another study we have found consistently high error rates associated with double-clicking [4].

Failing to follow instructions. In keeping with some of the observations made by Carroll and Mazur of people learning Lisa, we also found that people did not follow instructions as written. These included:

- Not reading every line of the instructions. If the missed line asked the user to change from Map Display to List Display, the Subject would not only be in the wrong display mode for the next exercise but would not be able to easily change directories, nor would the files be listed in the same way as for List Display. This simple error had a way of snowballing into something more serious.

- Typing in something from the description rather than from the exercises.
- Experimenting. Users would try things not mentioned explicitly in the instructions. In general, users should be encouraged to experiment with the system. However, when they are following a prescribed tutorial, such experimentation will likely result in the user getting to a place that is inconsistent with the tutorial.
- Clicking on the wrong directory to end up in an entirely different place than anticipated in the tutorial.

None of these are "wrong". However, they underscore some of the instructional difficulties observed by Carroll and Mazur and add a cautionary note to tutorial developers to beware of the many ways in which people may not follow the instructions exactly as given. The tutorial in this study was not written to be robust in the face of experimentation; by tying the instruction closely to the display it was intended to provide users with a consistent accurate feedback on their actions. A well produced tutorial would not be as restrictive nor as narrowly focused.

Timing. One of the most frustrating yet challenging features of teaching an application to new users is making sure users get the information they need *when they need it.* In the current study, we observed two examples where providing people with information was no guarantee of its use.

- *Paging Keys.* One of the first things Subjects were taught in the tutorial was how to use the scroll bar. They were also told that they could use the PgUp and PgDn keys on the keyboard. Most Subjects were delighted with this information, immediately experimented with these keys and pronounced them easier than the scroll bar. However, when it came to the remaining exercises, almost all Subjects went back to the scroll bar even though it was not comfortable to use. Why? We suspect that it was easier to learn (but not use) the mouse because, in almost all instances, the Subject was already using the mouse for other actions and switching to the keyboard would disrupt attention. Thus, even though the keyboard was thought to be easier, Subjects did not use it.
- *Visible feedback.* The interface was designed to provide people with feedback about status information. For instance, the name stripe at the top of the window told the user the current display mode; the top right of the file listings told users what page number was currently displayed in List Display mode; buttons were grayed out when they were not functional (i.e. when a file had not been selected), and the icons for directories were animated as they changed from a closed to an open version. Many of the less experienced Subjects did not notice the feedback until their attention was drawn to that part of the display by something else. For example, one Subject noticed the page numbers only when looking at a file that extended to the right hand side of the window and hence next to the page number. People who have not yet developed a mental model of the application may also not notice feedback because they cannot yet identify its significance. Feedback about user actions, for instance highlighting files when they are selected, was beneficial to users.

Active Learning. The training material fostered active learning by accompanying the description of each operation with an explicit script for the Subject to perform. This style of training was not only intrinsically rewarding but in certain cases, Subjects were able to use the "procedural" knowledge of how to do a task to compensate for their lack of conceptual knowledge. This was especially true for move or copy operations which had a very explicit set of commands but was conceptually complex. One of the Subjects who had never used a mouse before had some basic difficulties such as not clearly understanding the concept of a file or a directory but had no difficulty correctly completing the move and copy exercises in the "test" without assistance despite the conceptual difficulties inherent in those operations. In fact, there was some surprise amongst the software and interface designers at the acceptability by users of this method for moving/copying files. It had been thought that the explicit ordering of steps would prove cumbersome to users. Quite to the contrary the explicitness was desirable at least for novice users.

The main problem with the procedural approach is that it provides no help when the person takes a wrong path. One of the Subjects created a directory at the wrong level and had to move it to the next level up in the hierarchy. This proved to be somewhat difficult to accomplish because the Subject had to understand the hierarchical structure of the directories to figure out the target destination. After completing a move, the Subject said: "What is "practice"? Is it a file? "No, it is a directory, " I replied. "So they all moved into that?" Even after this dialogue, the Subject was not sure where the files were, nor what was the current directory.

The active learning approach works because it substitutes system response for interpretation of written material. For most actions there was immediate and visible feedback to the user in the form of highlighting a file or a button, an instruction popping up or an hourglass icon informing the user that something was happening. Even if the user does not understand how or why these commands work, the user can learn that a certain action is followed by a certain response on the part of the system. This method of learning is not only rewarding to the user but it is much simpler; "do this and then do this and then do this" requires much less interpretation or cognitive load than interpreting a description of how the system works. If users can build up a repertoire of simple procedural skills in this manner then they can begin to develop a knowledge base of the task domain. This knowledge base in turn can form the basis for acquiring new information by reading or experimentation.

Individual Differences. The procedural approach of the novices can be contrasted with the learning style of the few expert users in the study. Where novices ask for explanation and clarification, experienced users ask "what if.." and generate hypotheses. This frequently leads to the discovery of new information. The tutorial introduction deliberately omitted all information that was not essential to the task at hand. However, a few of the Subjects went beyond the information given by generating hypotheses and using their existing knowledge. One of the most experienced Subjects, a CAD user, was especially adept at this. For instance this Subject wanted a more direct way to get to a directory that was not in the current pathname. The Subject noticed the "open" button in the panel, and clicked on it to see whether that could provide a more direct route. In fact it did. In contrast, one of the inexperienced users also noticed the "Open" button. However this Subject mistakenly clicked on it to "open" the directory not understanding the display well enough to know that the directory was already open.

A necessary pre-requisite for generating useful hypotheses is having sufficient knowledge. In the case of more experienced users, that knowledge is often technical. For instance one person asked, pointing to the ancestor list after having changed directories, "Is that the actual path?" "Yes", I answered. "So it does a cd for you?" the person said using existing knowledge that "cd" means change directory in the Unix command language. This kind of specific technical knowledge lets users predict the behavior of the application thereby learning it faster and more thoroughly. Novices, on the other hand often stumble over basic concepts and terminology. For instance one of the most novice of the Subjects asked "What is a pathname", "What is an icon?" This person also didn't realize that the individual names in the pathname referred to separate directories. "Are these separate?" the Subject asked pointing to the list. "I thought they were like one long name." In contrast, the more experienced users had little or no trouble understanding the concepts and the elements of the application. One such Subject commented on one of the icons: "I know that's a directory because it has a folder."

These conjectures about a limited capacity for learning new information and the importance of active learning, if true, apply to many applications and interfaces. They suggest that the importance of making the interface easy to use for inexperienced users cannot be overestimated.

6. Discussion

The results of this study indicated that people liked Organizer and were able to use it successfully. However, we also uncovered several "bugs" as well as some fundamental problems in the learning rate of new users. Below we describe some of the changes made to Organizer before it was released as a result of the study.

Polishing the Application's Ease of Use

Based upon the feedback from the usability study many small modifications were made to the program which seemed trivial from a programmer's point of view but, together made a significant improvement in Organizer's ease of use and ease of learning. Changes included:

- Users could double-click to close as well as open directories in Map Display mode. Users were bothered that they could not close directories in Map Mode even though there was no need to close them since double clicking on another directory would close it automatically. This small feature gave users a better sense of control over the system.
- Users could update the display of the current directory by double-clicking on the last name in the pathname. "Update Display" technically does exactly the same thing but without truly confirming the files' existence to the user.
- To correct the problem of users trying to copy to a file, a notification window popped up after the user tried to "drop" to the file, telling the user that the destination for a copy must be a directory.
- Better feedback for Open, Rename, Delete, and Undo. In the original system, the files were re-positioned in sorted order when the user performed a Rename operation or a Delete operation. Many users found this jarring and would always search the directory for the new file name to make sure the file had been created or deleted. Create was changed to always put the new file or directory at the top of the directory listing. Rename was changed to leave the new file name in the same location as the original file before it was renamed. Delete was changed to leave a blank space where the file was originally positioned. If the user Undo-es any of these operations, the result appears at the original position of the file. The Update operation would re-sort and re-position the files on the display. Although the current system requires two steps to perform the operation that was originally performed in one step, it gives users more control of the system.
- The spacing between columns in Map Display was increased. This change reduced the mistaken selection of a calling directory in a copy operation which previously had resulted in an extended recursive copy.
- General "bugs" in the code were corrected and performance of the system overall was improved increasing the perceived reliability and responsiveness of the application.

Software engineering would have liked to incorporate more changes but it was not possible within the time frame. Moreover, changes such as the confusion between Map and List Display modes required additional research and prototyping before they could be reliably incorporated into the application.

Incorporating usability studies

The benefit of usability studies lies in the early and judicious application of the results to product development. In order to be effective, such studies must be undertaken early enough in the product development cycle to allow for feature changes to still be made. The inherent dilemma is that the software has to be sufficiently complete to be testable, but the development cycle should not be so far along that it is too late to make the changes. One solution is have long test cycles that permit feature changes. However, testing time frequently gets shortened in an effort to get a product to market early or because of earlier delays in development.

An alternative strategy is to develop early prototypes of the product that can be tested with users. Building these prototypes has been made easier by the advent of tools for prototyping user interfaces, commonly referred to as User Interface Management Systems (UIMS) (see Baecker & Buxton [1] for an excellent introduction to the research issues). Although traditional software development is moving towards a lifecycle that incorporates early prototype development, the effective involvement of usability testing and other human factors activities requires additional iterations of a prototype-test-develop sequence and additional development stages [9]. Despite the benefits in the quality of the product and its ease of use, usability testing is not yet common in product development [7]. This situation should change as UIMSs become commercially available. However, it also requires management and human factors professionals to work together to define the role and expectations that usability testing and other human factors activities can bring to the product and then to position those activities at the times where they can be most effective [8].

7. Summary

Testing the usability of a system can uncover problems which when pointed out make good sense but which might not have been predicted ahead of time. Examples from the current study include Subjects' use of the scroll bar rather than scroll keys even though the scroll keys were acknowledged to be easier, and, the failure amongst many users to notice status information such as the grayed out buttons. The study also exposed mismatches between user expectations and Organizer's behavior, and, problems distinguishing different display modes. Many of the mismatches were corrected before the product was released resulting in a more polished interface. We also observed some general trends. In particular, users liked our focus on active learning as a method for teaching a new application. Providing people with specific procedural steps for accomplishing simple tasks seemed especially beneficial for novice users who may lack the capacity for assimilating both new information about the interface and the application in a single session.

Acknowledgments

We would like to thank Matt Koehler for his careful reading and insightful comments on an earlier version of this paper.

8. References

[1] Baecker, R.M. & Buxton, W.A.S. Readings in Human-Computer Interaction. Los Altos, CA: Morgan Kaufman. 1987.

[2] Bewley, W.L., Roberts, T.L., Schroit,S. & Verplank, W.L. Human Factors Testing in the Design of Xerox's 8010 "Star" Office Workstation. In *Proceedings of CHI'83* (Boston, Dec.1983) 72-77.

[3] Callaghan, B. & Lyon, T. "The Automounter", *USENIX Conference Proceedings* (San Diego, Winter, 1989).

[4] Campagnoni, F.R. & Ehrlich, K . Information Retrieval Using a Hypertext-Based Help System. *Transaction on Office Information Systems* (In Press).

[5] Carroll, J.M. & Mazur, S.A. LisaLearning. *IEEE Computer 91(11)*, November 1986, 35-49.

[6] Carroll, J.M., Smith-Kerker, P.L., Ford, J.R. & Mazur-Rimetz, S.A. The Minimal Manual. *Human Computer Interaction* 3(2), 1987,123-153.

[7] Grimes, J., Ehrlich, K. & Vaske, J.J. User interface design: Are human factors principles used? *SIGCHI Bulletin 17*, 3 (1986) 22-26.

[8] Grudin, J., Ehrlich, S.F. & Shriner, R. Positioning human factors in the user interface development chain. In *Proceedings of CHI+GI'87* (Toronto, April 1987), 125-131.

[9] Mantei, M.M. & Teorey, T.J. Cost/beneft for incorporating human factors in the software lifecycle. *Comm. ACM 31*, 4 (1988) 428-439.

[10] Rubinstein, R. & Hersh, H. *The Human Factor*. Digital Press. 1984.

[11] Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, August 1983, 57-69.

[12] Shneiderman, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley:Reading, MA. 1987.

[13] Smith, D.C., Irby, C., Kimball, R., Verplank, W. & Harslem, E. Designing the Star User Interface. *Byte*, 7,4, April 1982, 242-282.

[14] Smith, S.L. and Mosier, J.N. *Guidelines for Designing User Interface Software*. ESD-TR-86-278, Mitre Corporation. 1986

[15] Stanley, B.L. "A Graphical Interface for the UNIX File System". Master's Project. Stanford University. 1984

[16] Valid "Allegro User Manual", Valid Logic Systems, Inc. 1987

The 3-D File System

David G. Korn
Eduardo Krell

AT&T Bell Laboratories
Murray Hill, NJ 07974

Traditional version control mechanisms in the UNIX® System have serious drawbacks. They work only on ASCII files and are encapsulated systems which do not fit nicely with the rest of the tools in the UNIX System. Sharing of files is not possible, as programmers have to make their own private copies when they want to work on different versions of a file.
This memorandum describes a powerful versioning mechanism which allows users to view different versions of files in a UNIX File System and allows multiple programmers to work on different versions of the software simultaneously, without having to make private copies of the files involved. This capability, now in prototype form, extends the concept of versioning to systems rather than files and to files with arbitrary format (e.g. object files and libraries) rather than only line oriented ASCII files (e.g. source code).
We have extended the concept of *viewpathing* in a way that is transparent to all programs without requiring any changes.
The current implementation is a user-level library which can be used to relink any program or command which is to understand versions and viewpathing. No changes need to be made to the user programs.

1 Introduction

The development of multi-release software systems has, for some time, been dependent on version control for multiple revisions of files of source code. In many cases, versions of documentation files are kept as well. As a software system evolves, there is a need to preserve its 'state' for a variety of reasons, as when developers have to work on different versions of a module simultaneously and must be able to precisely recreate prior versions of the system during maintenance.

We present a support mechanism for Version Control conceptually embedded within the UNIX System, rather than being built on top of it. This Version Control support mechanism allows for transparent access to different versions of a file, whether the file contains source or object code.

We also describe an extension of the two-dimensional UNIX File System into a third dimension. Here, one directory can be *overlaid* on top of another directory, making files of the underneath directory, whose names are not in the top level directory, appear as if they were part of the top level directory. This mechanism is an extension of the *viewpathing* concept, available until now only to specialized tools such as `build` [1] and `nmake` [2] .

One of the changes made to the UNIX System V kernel was the addition of a new file type called *version*. A version file is conceptually organized like a directory and is similar to hidden directories in the Locus [3] operating system. Whenever a user references a version file, a specific element within the directory is selected based on per-process information.

® UNIX is a registered trademark of AT&T.

This powerful mechanism provides the essential support for a user-level Version Control System which can be tailored to the needs of a specific software project.

These Version Control Systems are very easy to write due to the underlying services offered by our extensions to the UNIX System. We have implemented a prototype of such a system called VCS (Version Control System) as a set of shell functions.

Our objective is to provide an Integrated UNIX-based Software Development Environment (SDE) offering a consistent and logical view of the file system; we are especially interested in supporting development of multi-release software, where different versions of a system are being developed or maintained simultaneously, and the viewpathing model, where developers share the common, official sources and only keep local copies of the files they are changing.

2 Version Files

The Source Code Control System, SCCS [4] and/or the Revision Control System, RCS [5] are frequently used to keep track of multiple versions of a file. Both of these systems are limited to line oriented (e.g. source) files and cannot be used to keep versions of object or binary files.

Another drawback they have is their specialized user interface. Access to specific versions of a file is not transparent, and it requires construction of that version of the file. Furthermore, the constructed file does not have the attributes of the original file (e.g. time stamps, ownership, permissions). As a result, very few people use a system such as SCCS directly; instead, they use systems built on top of SCCS, which try to hide its awkward model and user interface.

Other UNIX System commands (like the compilers and editors) are unable to directly operate on SCCS and RCS files. Viewing a version of a file requires the additional step of creating it making it harder to use with existing tools. Viewing a different version of a system or a collection of files is even harder.

As an alternative, we introduced a new type of file to handle versions. A *version file* is conceptually organized like a directory but each time it is referenced, it returns a reference to to one of the files in this directory (these files are called *instances*). For example, a file **main.c** referred to by one user may resolve to a different instance for another user.

This is why we call it a transparent mechanism. No command is needed to construct a version of a file. Programs can operate on version files like if they were operating on regular files.

Figure 1 shows a version file **x.c** with two instances: **1.1** and **1.2**. The difference between **x.c** and a directory is that when a reference to the name **x.c** is made, the system will select one of its instances. Additionally, specific instances can be selected as **x.c/1.1** and **x.c/1.2**.

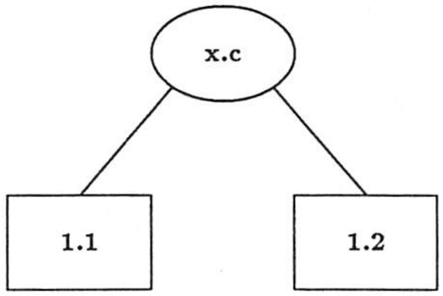


Figure 1: A Version File

It should be pointed out that the names **1.1** and **1.2** are arbitrary names and we could have used any other valid file name instead. Numeric names are useful for a software configuration control tool to show the chronological relationship among instances. Mnemonic names are easier to remember and can be used to hold together related instances (like all the instances that make a specific release of a system).

An instance of a version file can have more than one name through the link facility in the UNIX System. We can link the **1.1** instance with the name **generic1** which would make **generic1** synonymous with the **1.1** instance in that file. Figure 2 shows the same version file as Figure 1, only one of the two instances has an alias. The use of this facility will be further explored in section 5.2.

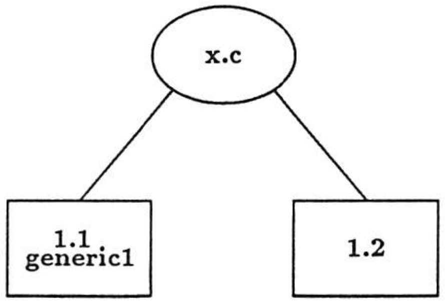


Figure 2: Multiple named instances

The system has to examine the last component of a file name to see if it is a version file and compute the corresponding version instance. Each process should be able to control the selection of the instance given the name of a version file. The use of environment variables was not possible from the kernel since the environment is stored in user space and is susceptible to being corrupted by a user process. Instead, we added new system calls to control version mapping.

It is sometimes desirable to look at the version object itself, rather than a specific instance. We use a trailing / to convert a version file into a directory. Thus, if **main** is a version file, then **main/** is the directory which contains all the instances of **main**. Each instance can be named explicitly by **main/***instance*. In this respect, **main/** behaves as an ordinary UNIX System directory.

We do not allow directories to be versioned objects, since the same effect can be achieved using a viewpath (see section 4). However, a version instance **can** be a versioned object. This feature is used to create branches of changes to previously released code.

3 Version Mapping

In order to select which instance should be chosen when it encounters a version file, each process keeps a table specifying the rules of instance selection. This table is inherited by a child process from its parent. Each element of the table contains a reference to a directory and an list of instance names separated by slashes. The meaning of such an entry is that whenever a reference to a version file in that directory (or a subdirectory thereof) is made, the corresponding list of instance names is searched (in order) until a match occurs, if any.

Table 1 and Figure 3 show a subtree of a UNIX file system and a version mapping table that applies to it. Its meaning is as follows: for all version files under **/usr**, the **1.1** instance should be selected. For all version files under **/usr/gsf/io**, the **1.2** instance should be selected, thus overriding the **1.1** selection applicable under **/usr**. Finally, version files under **/usr/dgk/db** are mapped to the **1.3** instance, or the **1.2** instance if there is no **1.3**.

When a version file is encountered, the instance selection proceeds as follows:

Directory	Version
/usr	1.1
/usr/gsf/io	1.2
/usr/dgk/db	1.3/1.2

Table 1: Version Mapping Table

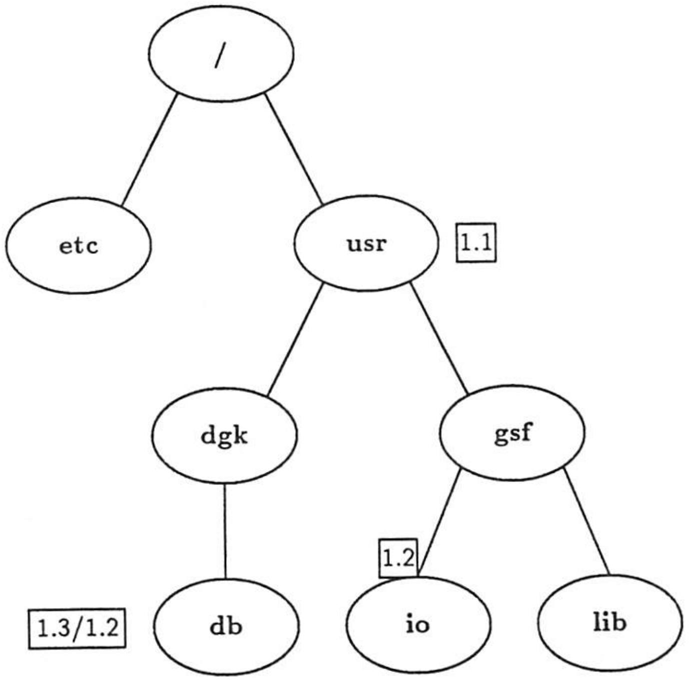


Figure 3: Version Mapping

- Starting with the current directory and moving up towards the root, if an entry in the version mapping table is found for that directory and one of the instances in the corresponding list exists, then select it.
- If the above search is exhausted and a **.default** instance exists, select it.
- Else, return an error (File does not exist).

4 The Viewpathing Concept

The model behind viewpathing is that of a software system where "snapshots" have been taken at key points during its life cycle. In the simplest case, a developer is changing some files in a system (to fix a problem or make an enhancement, for instance). The system sources reside in an "official area" (**/ofc** in Figure 4). In the example illustrated in Figure 4, these sources are distributed in three subdirectories: **src**, **inc** and **lib**.

The developer is not allowed to change the files in the official area, since other developers depend on those files not changing, and they might not be interested at all in the changes this developer is making.

A developer copies files that require change to their own directories (**/dvl**) and work there. Figure 4 shows this situation. The developer has copied and changed 2 files (x.c in the **src** directory and x.h in the **inc** directory). We use a different font (like this) to visually identify those files in the developer's directory.

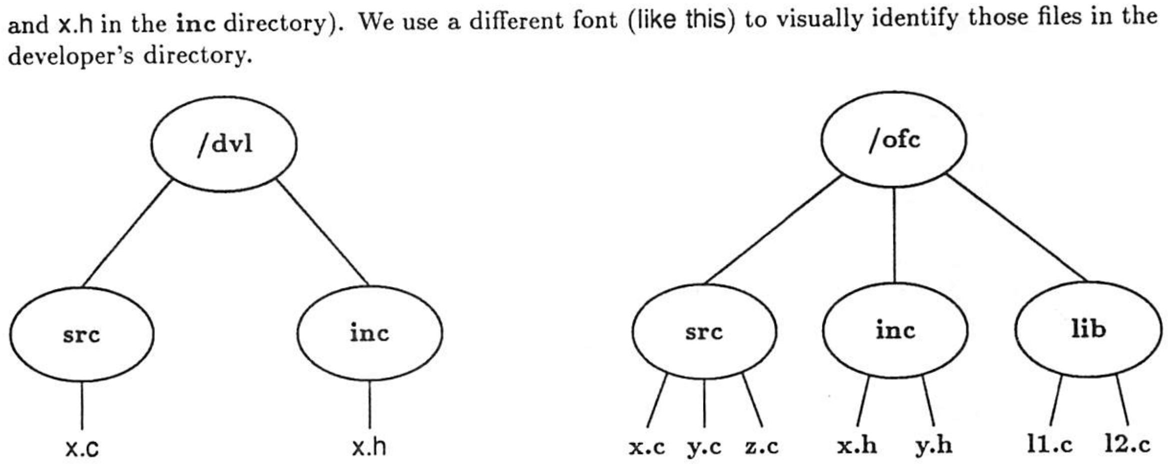


Figure 4: Official Source

When it comes the time to rebuild the system, if viewpathing is not used, the files in the official area *which have not been changed* are manually copied to the developer's directory (or linked if they are on the same file system), as shown in Figure 5, and then the system is rebuilt. This copying process can involve large amounts of data and time, and if several developers are all trying to do it simultaneously, there might not be enough disk space to hold multiple copies of the sources.

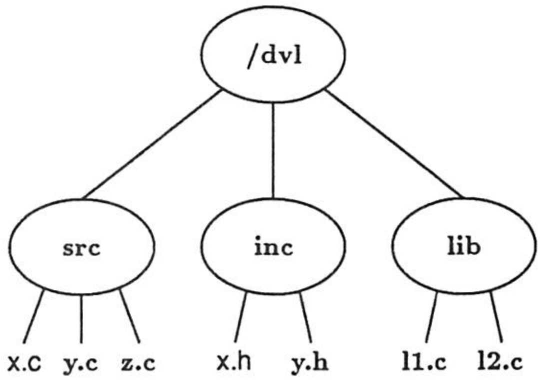


Figure 5: Merged Source

The concept of Viewpathing first appeared in `build` [1] and is used by **`nmake`** [2], two successors to the UNIX System program `make`, a configuration management tool used to rebuild a system when some of its components are changed.

Build uses the VPATH environment variable to search for source files required by the makefile and automatically links or copies these files to the developer's node. Build removes these source files after they are compiled. One major problem with this approach is that the file system may be left in an inconsistent state, in which some official sources were copied to the developer's directory for compilation but were not removed, when build terminates prematurely. The developer is then left with the problem of identifying and removing those files. In addition, it becomes impossible to rely on time stamps on the source files since they get copied every time the system is rebuilt, and so they get new time stamps each time.

`Nmake` avoid this problem by using the VPATH variable to generate the pathnames for source files and –I switches for the header files. For instance, in our example, **`nmake`** will use the files x.c and x.h from the developer's directories, but it will use the rest of the files from the official area, with their full pathnames.

Although to `nmake` and `build` the system being built looks like the one in Figure 5, it is actually only an abstraction inside them, and does not really exist as such. For instance, the developer could not look at the file **y.c** in the **src** directory because that file does not exist in the developer's directory, but rather is in the official area only.

Multiple directory trees add to the complexity. For instance, after the system has been released, it goes through a System Test phase, where errors are found and fixed. These fixes are not applied to the official source. Instead, they are kept in a separate area. This can be repeated over and over and we have seen middle-size projects with seven or more levels of nested source areas. In such an environment, it is very hard for a developer to find out exactly where the source for a given file resides.

4.1 Transparent Viewpathing

We believe the concept of viewpathing is important and that it should be an integral part of our SDE, available to all tools, not just a few. This will provide a consistent, logical view of the system being built. We added viewpaths to the UNIX System by building a Viewpath Table, which contains all the viewpaths the user wants to set.

In the example above, the Viewpath Table would contain an entry like

Directory	Viewpath
...	...
/dvl	/ofc

Table 2: Viewpathing Table

Any UNIX System tool will see the **/dvl** subtree as the one in Figure 5.

It is important that all system calls which take pathnames as arguments behave consistently with the viewpathing concept. This requires all such system calls to search the different levels in the viewpath to resolve a pathname reference.

Directories must be made to appear as the union of the top level and all the underneath directories. Since the interface for directory access is localized via the `opendir()`, `readdir()`, etc. routines, changing these functions is all that is needed to provide this abstraction to higher level programs. When `readdir()` gets to the end of the current directory, it will open the corresponding directory in the next accessible level in the viewpath. It is necessary to remember the names of the entries on previous directory levels to avoid returning the same entry name twice.

4.2 The Third Dimension

We look at the File System as having two dimensions, breadth and depth. The breadth is given by "navigating" in a given directory and the depth is given by navigating through the file system using subdirectories to go one level down and `..` to go one level up.

We want to think of the `..` name as a navigational operator whose meaning is to move one step closer to the root, which is consistent with the way most people think of it.

Unfortunately, the implementation of symbolic links in the Berkeley version of the UNIX System broke this paradigm. For instance, you can have a symbolic link called `/usr/include/sys` to a directory whose real pathname is `/sys/h`. When you do a `cd /usr/include/sys`, the name ".." refers to `/sys` and not to `/usr/include`, which is quite disturbing. The reason for this interpretation is that ".." is actually a hard link in `/sys/h` pointing to its parent directory, `/sys`.

We want to interpret the meaning of ".." as being a *logical parent directory* operator, which refers to the parent directory of the current working directory by looking at the pathname used to get to it. So, if you use the name `/usr/include/sys` to get to that directory, the name ".." will always refer to `/usr/include`, whether `/usr/include/sys` is a symbolic link or not.

The third dimension is added by the establishment of viewpaths. A directory becomes the union of itself and a number of other directories. It can also be viewed as a series of directories stacked one on top of the other. Hence, the third dimension.

We also need an operator to navigate this third dimension. We called that operator "...". If there are any directories stacked under a given directory, the name "..." will refer to them. For instance, you can execute

```
ls ...
```

to see all the files in the underlying viewpath(s). A viewpath can be seen as a linked list of file system subtrees and the "..." operator removes the top level subtree.

To better understand the use of the name "...", let's look at an example. A typical usage would be to get to the viewpath to see the files that would be obscured or hidden otherwise. If you have a file `cmds.c` which is in both the top level directory and some underneath directory, the top level version of `cmds.c` hides the one in the viewpath directory. The name `.../cmds.c` refers to the file `cmds.c` binds to after the top level directory is **removed** from the viewpath. Whenever `cmds.c` and `.../cmds.c` refer to the same file, `cmds.c` is not in the top level directory.

Thus, to see the differences between the top level `cmds.c` and the one in the viewpath, you can just type

```
diff .../cmds.c cmds.c
```

Here, the name `.../cmds.c` means "remove the top level from the viewpath and find the file `cmds.c`". The file does not have to be in the second viewpath level. If the viewpath is more than two levels deep, `.../cmds.c` might actually be in the third or the fourth level. In a similar fashion, `.../.../cmds.c` means "remove the two top level subtrees from the viewpath and find the file `cmds.c`". It does not make sense to refer to `.../.../file` if the viewpath is less than 3 levels deep, since that file cannot exist.

4.3 Read-Only Viewpath

We decided to treat all the files and directories not in the top level as read-only, regardless of actual file system permissions. It is as though the underneath directory had been mounted read only. The files can still be modified by explicitly referring to them with full pathnames or via the ... mechanism.

Files are never created or removed from underneath directories (unless explicitly referenced with full pathnames). We prevent this by not allowing the implicit search of underneath directories for the `creat()`, `mkdir()`, `rmdir()` and `unlink()` system calls. In the case of `unlink()`, we do not report an error if there is an underneath file so that commands like `rm *` at the shell level remove top level files but do not report errors for files in underneath directories that are not removed.

Fortunately, this works with virtually all UNIX System programs. For instance, an editor like `vi` reads the file being edited into its buffer and then it creates the file before writing it back. If the file being edited is not in the top level, `vi` will read the file from some underneath directory and write it to a new file in the top level directory. Other programs, such as `ar`, behave in a similar way. One of the few programs that required changes was `cp`. In particular, we wanted to allow `cp file file` or `cp file .` to work when `file` was in an underneath directory. We fixed `cp` at the point where it discovers that the two files are identical and if it was an underneath file we allowed it to continue.

Changing directory with the `chdir()` system call or the `cd` shell builtin does not create directories. The system maintains the logical name of directories (the name used to get to a directory via `chdir()s`) and it reports that name as the name of the current working directory, whether the directory is a real or virtual one.

A virtual directory is one which does not exist in the file system, but logically exists when viewpathing is applied. Consider a viewpath from `/user1` to `/user2` where `/user1` is initially empty and `/user2` has a subdirectory called `src`. You can change directory to `/user1/src`, even though there is no such directory. `/user1/src` is a virtual directory.

When a request for creating a file or directory in a virtual directory is made, instead of returning an error message, we create a real top level directory (`/user1/src` in our example) on the fly and change directory to it. The file or directory will then be created in this newly created top level directory. Sometimes creating a file can involve creating a chain of directories (if you were in `/user1/src/lib`, for instance).

5 User Level Interface

5.1 User Level Primitives

At the user level, the commands **mkver** and **rmver** were added to create and remove version files. **mkver** takes a regular file as an argument and converts it into a version file and places the original file as the **.default** instance. **rmver** is the inverse: it takes a version file with only one instance and converts it back into a regular file.

We also added a pair of new builtin commands to the korn shell [6] , **vmap** and **vpath**, used to list, add, or remove entries from both the version mapping and the viewpathing table.

5.2 User Level Version Control

On top of these commands, a prototype of a version control system named, appropriately, VCS, has been built. VCS is an example of the power of the underlying version files facilities. The goals behind VCS are to provide controlled access to version files and to preserve an audit trail of changes made to version files.

It is implemented as a set of shell functions and it was coded in an afternoon. A more sophisticated, *production quality* replacement can be written with very little extra effort.

VCS offers the following subcommands:

create Converts a plain file into a version file.

checkin Checks in a new instance of a version file that has been previously checked out.

checkout Checks out an instance of a version file for editing.

delete Deletes an instance or an entire version file.

log Appends an entry to the log file. Used by **checkin** to create an audit trail of changes made to this file.

product Links specific instances of version files under a common name. Used to give a symbolic name like **generic1** to a collection of version files instances.

5.3 Concurrent Update Detection

When a user checks out an instance of a version file for update, VCS writes an entry in the log file to make a record of that event. VCS will not allow a second checkout for update as long as that version hasn't been checked back in (or the original checkout been cancelled). The user who is attempting to check that file out will get an error message from VCS, which will include the instance of the file that has been previously checked out and the user who checked it out.

This is one example of possible audit trail mechanisms which could be built using our system.

5.4 User Level Viewpathing

Two builtin commands were added to the UNIX System shell: **vmap** and **vpath** to manipulate the Version Mapping and the Viewpathing tables, respectively. In our examples, the Version Mapping table of Figure 3 listed in Table 1 can be created by the user issuing the following sequence of commands:

```
$ vmap /usr 1.1
$ vmap /usr/gsf/io 1.2
$ vmap /usr/dgk/db 1.3/1.2
```

The dollar sign and the space following it are printed by the shell as its input prompt. The user's input follows "`$ `". Similarly, the Viewpath described in Section 4 is created by:

```
$ vpath /dvl /ofc
```

Both **vmap** and **vpath** can be used to display the respective tables when no arguments are supplied and they can be used to remove an existing entry in the tables by supplying a second null argument.

6 Implementation Status

An earlier version of the system described in this memorandum was implemented by changing the UNIX System V Release 3 kernel. The current implementation is built entirely at the user-level, and requires no changes to the kernel. We provide a library which replaces all the functions in the C Library and the system calls which take pathnames as arguments with our own. Any program linked with these libraries will have access to both versioning and viewpathing as described above without having to be modified.

This implementation runs slower than the kernel one and it requires relinkage of the binaries whereas the kernel implementation does not require even that, but it is portable and can be used by an individual or a group without disturbing the rest of the user community (a request for a kernel change would not be well received by system administrators). We also provide the changes to the Korn Shell required to add support for versioning and viewpathing.

7 Future Work

We are interested in building an experimental software configuration management tool based on these concepts and integrate it with `nmake` [2] , an improved version of the standard UNIX System `make` tool.

We are also investigating compaction methods which would allow the individual instances to be compacted into one physical file, storing only the *deltas* or differences between instances.

References

[1] V. B. Erickson and J. F. Pellegrin. Build A Software Construction Tool. *AT&T Bell Laboratories Technical Journal*, 63(6):1049–1059, July 1984.

[2] Glenn S. Fowler. The Fourth Generation Make. In *USENIX Portland 1985 Summer Conference Proceedings*, pages 159–174, 1985.

[3] G. Popek and B. Walker. *The LOCUS Distributed System Architecture.* MIT Press, 1985.

[4] L. E. Bonnani and C. A. Salemi. Source Code Control System User's Guide. In *System V Programmer's Manual.* AT&T.

[5] Walter F. Tichy. RCS - A System for Version Control. *Software Practice & Experience*, 15(7):637–654, 1985.

[6] M. Bolsky and D. Korn. *The Korn Shell Command and Programming Language.* Prentice Hall, 1989.

The C Program Database and Its Applications

Yih-Farn Chen

AT&T Bell Laboratories
Murray Hill, NJ 07974

The C program database is a collection of files that stores the structure information about software objects in C programs. These files can be processed by relational database tools to locate object declarations and analyze their relationships. This paper describes how various tools use the C program database to provide graphic views of program structures, extract self-contained subsystems, eliminate dead code, and automate software restructuring tasks. Program databases have been built for several widely used tools and systems. The analysis results of these databases are compared.

1 Introduction

Understanding program structures is a time-consuming process in software development and maintenance. A programmer's productivity is strongly affected by the ability to locate software objects and analyze the relationships among them.

A traditional solution to this problem has been cross-reference tools, such as *Masterscope* [1] in the Interlisp environment and *Cscope* [2] in the C programming environment. The problem with these tools is their extensibility. Because of the lack of a suitable database schema, the reference information is usually kept in a form that cannot be easily processed by other tools.

Omega [3] uses the INGRES [4] relational database system to manage the program database information for a Pascal-like language called *Model*. The advantage of *Omega* over *Masterscope* or other similar tools is that it can fully utilize the query power provided by INGRES. Unfortunately, due to its complex database schema (fifty-eight relations) for describing all the detailed relationships between program elements, the prototype implementation of OMEGA has poor response time in retrieving the body of a procedure.

The C Information Abstractor [5] , *cia*, is a tool that extracts relational information from C programs and stores that information in a database. Unlike *Omega*, however, a concise conceptual model that focuses on *global* objects (objects that can be referred to across function boundaries) is used to guide the extraction process of *cia*. The model is rich enough that sophisticated C tools can be easily built on top of the program database, yet it is simple enough to render the resulting database manageable.

This paper explains how various tools use the C program database to chart program structures, extract self-contained software components, and eliminate dead code. *Cia* and all the tools built on top of the C program database are collectively called the C Information Abstraction system (CIA)[1]. CIA is not only useful in keeping track of one's own C programs, but is also helpful in understanding and manipulating programs written by others.

Section 2 gives an overview of the CIA system. Section 3 describes the CIA conceptual model, which defines the objects and relationships to be extracted by *cia*. Section 4 gives a simple tutorial

[1] In this paper, we use the lower case italic word *cia* to denote the C Information Abstractor, and the word CIA in capital letters to denote the C Information Abstraction System.

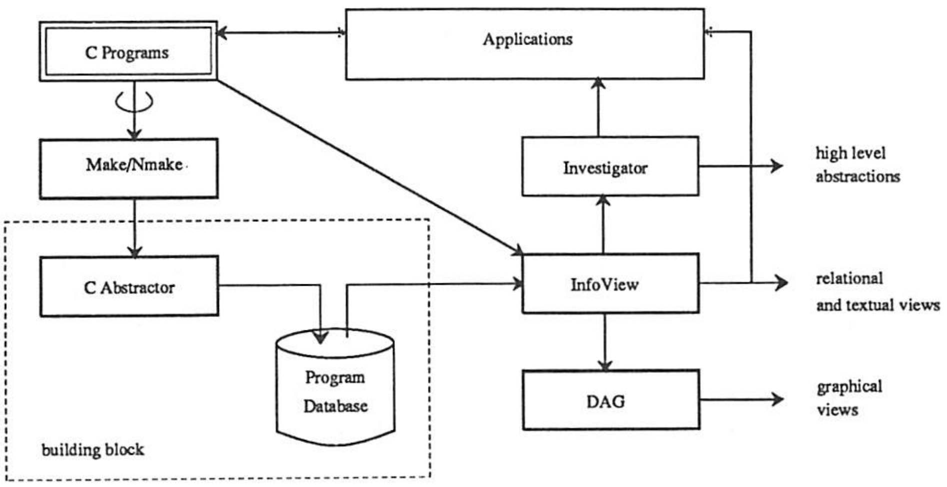


Figure 1: The C Information Abstraction System

on program database queries. Section 5 describes how the C program database is used to generate graphical views. Section 6 demonstrates the use of CIA in exploring various aspects of program structures. Section 7 shows how to maintain a large program database and compares several program databases. Finally, Section 8 gives the current status of CIA.

2 System Overview

The C program database interfaces nicely with tools that run on UNIX® systems. Figure 1 shows how these related tools interact with each other. Normally, *make* or *nmake* [6] is used to check the time stamps of C source files and then create or update a C program database by invoking *cia*.

The relational information stored in the database can be processed by *awk* [7] and a variety of database retrieval systems, including INGRES and the Information Viewer(InfoView), a system specifically designed for accessing the C program database. InfoView provides relational views to users or other application programs.

DAG [8] is used to draw directed graphs that show several aspects of program structures, including function call graphs, data structure maps, and file include hierarchies.

The *Investigator* is a collection of tools that process the relational views to provide some interesting software abstractions such as software layers, cross couplings, and reachable sets.

Cia and the C program database comprise an ideal building block for developing C related tools. They eliminate the need for parsing C programs to extract the structure information that many C tools require. More tools and applications that interface with the C program database or other components in the CIA system are being built.

The current version of *cia* accepts C programs that conform to either the draft proposed ANSI C standard [9, 10] or the language defined in the first edition of *The C Programming Language* [11].

® UNIX is a registered trademark of AT&T.

```
<trans.c>
#include "coor.h"
#define FNUM 2
#define DELTA(x) (x + 1/x)

extern COOR *rotate();
extern COOR *shift();
typedef COOR *(*PFPC)();

static PFPC ftable[FNUM] =  { rotate, shift } ;
int tsize = sizeof(ftable) / sizeof(PFPC);

trans(angle)
int angle;
{
  ftable[0](angle);
  shift(DELTA(angle));
  return(tsize * sizeof(COOR));
}
```

```
<op.c>

#include "coor.h"

COOR *rotate(degree)
int degree;
{  /* ... */  }

COOR *shift(distance)
int distance;
{  /* ... */ }
```

```
<coor.h>

#define DIM 2
typedef struct coor COOR;
struct coor {
    int point[DIM];
    COOR *next;
};
```

Figure 2: A Textual View of **Trans**

3 The CIA Conceptual Model

The CIA conceptual model defines the software objects, attributes, and relationships to be stored in the C program database. It serves as a requirements specification for *cia*. It also limits the knowledge that all CIA related tools can provide to users.

3.1 Object Kinds

Cia records information about five kinds of *global objects* in C programs[2]: files, macros, global variables, types (including typedefs, structures, and unions), and functions. An object is global if its identifier can be referenced across boundaries of external declarations[3] in C programs. We refer to these objects as CIA objects.

Figure 2 shows a simple C program **Trans** that consists of fourteen CIA objects:

[2] The definition of *object* here is different from the one used in the Draft-Proposed ANSI C standard. Other than that, the terminology used in this paper follows the ANSI C standard.

[3] Note that a function definition is also an external declaration.

`file:`	`trans.c, coor.h, op.c`
`macro:`	`DIM, FNUM, DELTA`
`type:`	`"struct coor", COOR, PFPC`
`global variable:`	`ftable, tsize`
`function:`	`rotate, shift, trans`

3.2 Attributes

A CIA object can have several declarations. Each declaration has a set of attributes, which describe its location and other useful information. For example, the declaration of the static variable **`ftable`** shown in Figure 2 resides in **`trans.c`**, has the data type `PFPC`, starts at line 9 and ends at line 9. Therefore, the attribute list of **`ftable`** is as follows:

`file:`	`trans.c`
`data type:`	`PFPC`
`name:`	`ftable`
`storage class:`	`static`
`beginning line:`	`9`
`ending line:`	`9`

Each object kind has a similar attribute list. The CIA conceptual model does not explicitly define the order or organization of the attributes stored in the database, but leaves this to the underlying relational database schema.

3.3 Relationships

Cia records *reference* relationships between CIA objects. A reference relationship exists between objects A and B if one of the following conditions holds:

- If A is not a file and an external declaration of A refers to B. The reference can occur before, during, or after C preprocessing.
- If A is a file, then A refers to B if B is a file included by A.

In other words, if A refers to B, then the compilation of A or the subsequent linking of the compiled code requires processing the definition of B to make an executable program. Table 1 lists all the reference relationships recorded by *cia*.

As an example, the following reference relationships can be derived from the program **`Trans`** shown in Figure 2: [4]

object 1	relationship	object 2	comment
trans.c	includes	coor.h	file to file
PFPC	refers to	COOR	type to type
ftable	refers to	PFPC	gbvar to type
ftable	refers to	FNUM	gbvar to macro
ftable	refers to	rotate	gbvar to function
ftable	refers to	shift	gbvar to function

[4] As an exercise, construct a C program that has a type to gbvar relationship (it rarely occurs in real C programs).

num	*objkind1*	*objkind2*	*definition*
1	file	file	file1 includes file2
2	function	function	function1 refers to function2
3	function	gbvar	function refers to gbvar
4	function	macro	function refers to macro
5	function	type	function refers to type
6	gbvar	function	gbvar refers to function
7	gbvar	gbvar	gbvar1 refers to gbvar2
8	gbvar	type	gbvar refers to type
9	gbvar	macro	gbvar refers to macro
10	type	gbvar	type refers to gbvar
11	type	type	type1 refers to type2
12	type	macro	type refers to macro

Table 1: Reference Relationships in C Programs

```
tsize           refers to     ftable         gbvar to gbvar
tsize           refers to     PFPC           gbvar to type
trans           refers to     ftable         function to gbvar
trans           refers to     shift          function to function
trans           refers to     DELTA          function to macro
trans           refers to     COOR           function to type
trans           refers to     tsize          function to gbvar
COOR            refers to     struct coor    type to type
struct coor     refers to     COOR           type to type
struct coor     refers to     DIM            type to macro
op.c            includes      coor.h         file to file
rotate          refers to     COOR           function to type
shift           refers to     COOR           function to type
```

These reference relationships are shown in a directed graph in Figure 3, in which functions and files are shown in boxes, while types, macros, and global variables are shown in ovals. The picture shows that reference relationships in a small program like Trans can have as many as seven levels. It is difficult to realize this without the help of the program database and the graphical view. In Section 5, we will give more details about how various graphical views can be constructed automatically from the program database.

Besides recording the relationships listed in Table 1, the line number of each reference is also recorded in the program database[5].

4 Relational Views: The InfoView System

The Information Viewer (InfoView) is a collection of tools for retrieving information from the program database created by *cia*. The command syntax of *cia* is very similar to *cc* (Section 7 gives more details on this). For example, the following command creates a program database for the set of C source files shown in Figure 2:

```
$ cia trans.c op.c
```

[5] In the case of functions, two line numbers are kept: the beginning and ending line of the reference. This is because a function call may span several lines.

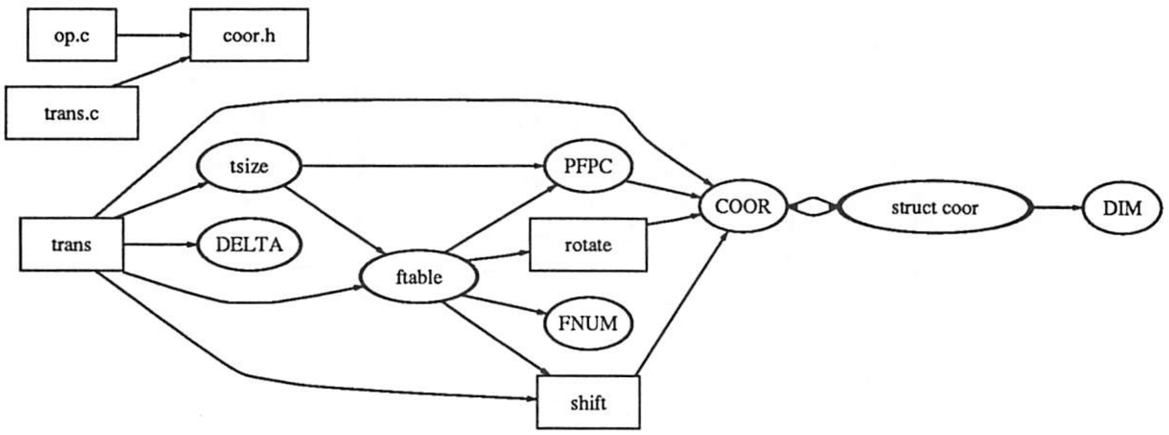


Figure 3: A Graphical View of the Reference Relationships in **Trans**

In this and all the following examples, a shell prompt **$** is shown before each command line, which is followed by its output.

There are four basic commands in InfoView:

```
def: display the attributes of object definitions or declarations
ref: display the relationships between objects
viewdef: view the text of object definitions or declarations
viewref: view the text of object references
```

These commands are based on the concepts of objects, attributes, and relationships. A user needs to understand only the CIA conceptual model to use most of the InfoView commands. A knowledge of the underlying relational database schema or a complex query language is not required.

4.1 Def - Get the Attributes of Object Definitions or Declarations

Def gives information about the attributes of object definitions or declarations. An object is specified by its object kind and name. The command syntax is

```
def [options] obj_kind obj_name [selection_clauses]
```

For example, to see the attributes of the function **trans**, type

```
$ def function trans
file               sclass dtype             function          bline eline
================== ====== ================= ================= ===== =====
trans.c            global int               trans             12    18
```

The output shows that the function definition of **trans** is defined in the file **trans.c**, returns an integer, begins at line 12, and ends at line 18[6].

InfoView commands recognize the minus sign, "-", as a *wildcard*, i.e., it matches any symbol and any object kind. For example, you can get a list of all the function declarations by specifying the

[6]The **sclass** field of a non-static function or global variable is set to **global**.

object name as "-". Regular expressions similar to that used in *egrep* are also acceptable argument names.

The output can be further restricted by specifying a set of *selection clauses* in the form of *fieldname=fieldvalue*. For example, to get a list of all function declarations that return a pointer to the data type COOR, use

```
$ def function - dtype="COOR *"
file              sclass dtype              function          bline eline
================= ====== ================== ================= ===== =====
trans.c           extern COOR *             shift             6     6
trans.c           extern COOR *             rotate            5     5
op.c              global COOR *             rotate            3     5
op.c              global COOR *             shift             7     9
```

Similar queries can be applied to other kinds of CIA objects. For example, it is easy to write a shell script that uses *def* to detect multiple definitions of the same macro.

Both *def* and *ref* allow the user to specify a `-u` option to output the data in an unformatted form so that it is more suitable for processing by other commands. The colon character ":" is used as the field separator. For example, to get the unformatted form of the attribute information about the function `trans`, use

```
$ def -u function trans
5:trans:function:trans.c:global:int:12:13:18:0:0
```

The `file` and `bline` values can be used by a shell script to invoke an editor that places the cursor on the beginning line of any object declaration. However, if a source file is modified, then some information in the program database may become out of date. Section 7 shows how to update a program database incrementally.

4.2 Ref - Display Relationships Between Objects

Ref retrieves the relational information between two kinds of objects. The command syntax is

```
ref [options] obj_kind1 obj_name1 obj_kind2 obj_name2 [selection_clauses]
```

The first and second arguments specify the parent of the reference relationship and the third and fourth arguments specify the child. Any argument can be specified as a wildcard. For example, to show all the objects that refer to the function `shift`, use

```
$ ref - - function shift
file1        kind1 name1             file2        function
============ ===== ================= ============ =================
trans.c      gbvar ftable            op.c         shift
trans.c      funct trans             op.c         shift
```

It shows that `shift` is referred to by the global variable `ftable` and the function `trans`.

Optional selection clauses are also available for the *ref* command. The following command retrieves all the references to the data type `COOR` made in `trans.c`.

```
$ ref function - type COOR file1=trans.c
file1              function           file2              type
================== ================== ================== ===================
trans.c            trans              coor.h             COOR
trans.c            shift              coor.h             COOR
trans.c            rotate             coor.h             COOR
```

It is easy to compute certain software metrics using the *ref* command. For example, the following command counts the number of cross references between objects in the files **trans.c** and **op.c** (you might want to find these references in Figure 2):

```
$ ref -u - - - - file1=trans.c file2=op.c | wc -l
       3
```

4.3 Viewdef - View the Definition of an Object

Viewdef prints out the definition or declaration of a specified object. The syntax is very similar to that of *def*:

```
viewdef [options] obj_kind obj_name [selection_clauses]
```

For example, to see the definition of the data type COOR, type the following:

```
$ viewdef type COOR
typedef struct coor COOR;
```

Now you might be interested in the definition of `struct coor` and then the definition of **DIM**:

```
$ viewdef type 'struct coor'
struct coor {
    int point[DIM];
    COOR *next;
};
$ viewdef -f -n macro DIM
coor.h:1     #define DIM 2
```

The optional argument **-f** allows file names to be printed. The other argument **-n** allows line numbers to be printed. *Viewdef* allows you to trace the definitions of objects on any reference path easily.

4.4 Viewref - View the Text of References to Objects

Viewref prints out the text of actual references to objects. The syntax is very similar to that of *ref*:

```
viewref [options] obj_kind1 obj_name1 obj_kind2 obj_name2 [selection_clauses]
```

For example, to see all the references to **shift**, type

```
$ viewref -f -n - - function shift
trans.c:9     static PFPC ftable[FNUM] =  { rotate, shift };
trans.c:16      shift(DELTA(angle));
```

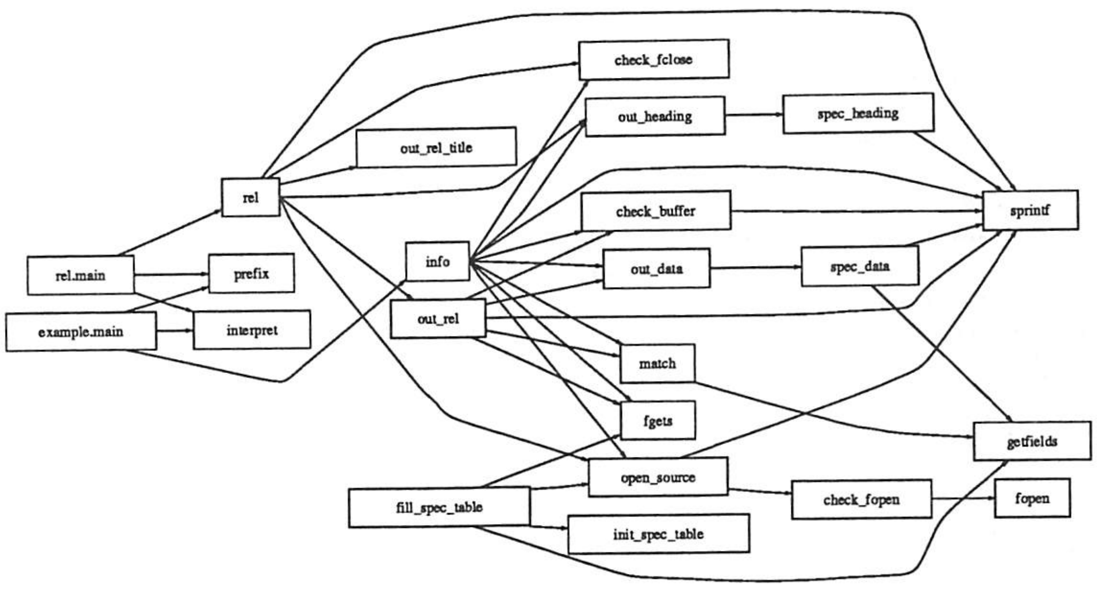


Figure 4: A Function Call Graph

5 Graphical Views

Information stored in the program database is often used to generate pictures that show interesting aspects of the internal structure of C programs. The tool *dagen* extracts relationships in the database to generate picture descriptions that are processed by *dag*, a tool for drawing directed graphs [8] . The command syntax of *dagen* is

```
$ dagen obj_kind1 obj_kind2
```

This section shows three types of graphs that are commonly generated: function call graph, file include hierarchy, and data structure map.

A function call graph description in the DAG format can be generated by the following command after the program database is built ("$PRINT" is to be replaced by the local command used to print a PostScript picture.):

```
$ dagen function function | dag -Tps | $PRINT
```

Figure 4 shows the function call graph of a small program. The picture layout was done automatically by *dag* without any human intervention. It shows clearly the function layering structure and the fan-ins and fan-outs of each function.

Figure 5 shows the file include hierarchy in another program. It is simply a graphical view of the file to file relationship. The picture was printed by the following command:

```
$ dagen file file | dag -Tps | $PRINT
```

The picture shows that seven header files must be processed in order to compile `pathmap.c`.

Figure 6 shows a set of data structures and their relationships in a library. The picture was printed by the following command (the `-R` option rotates the picture by 90 dgrees):

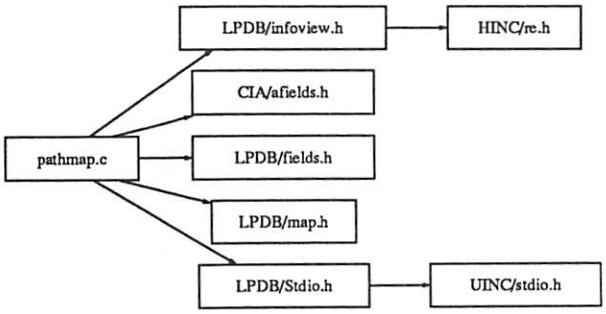


Figure 5: A File Hierarchy

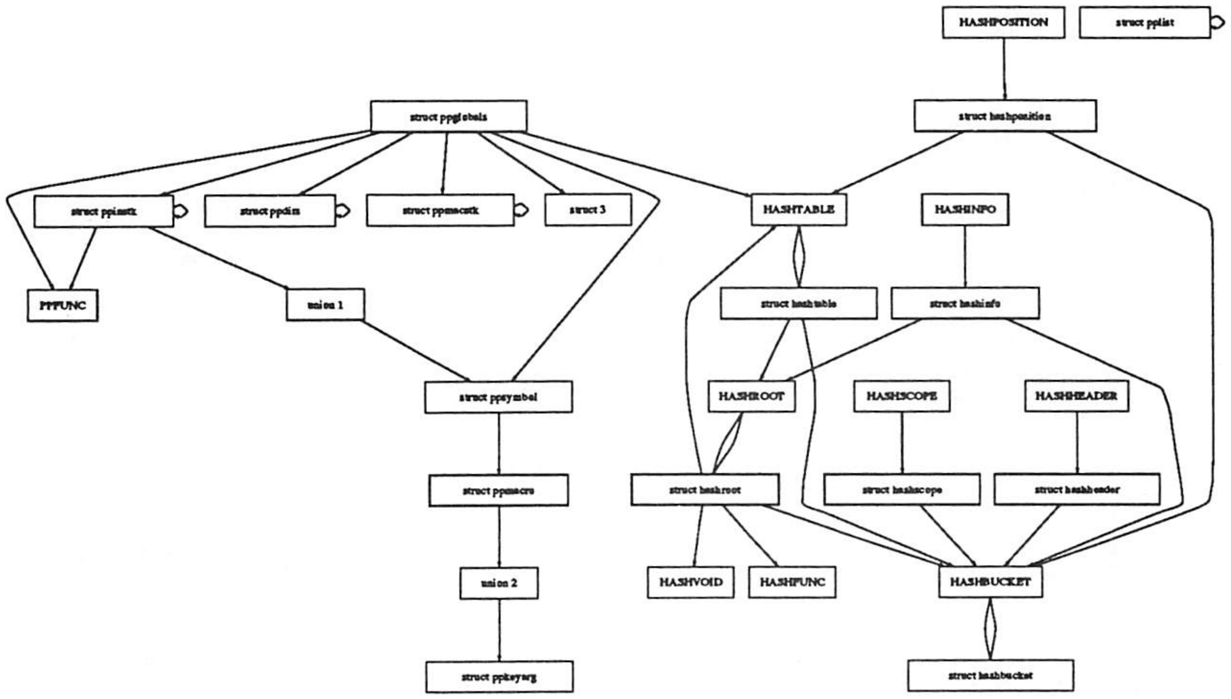


Figure 6: The Data Structure Map for libx.a and libpp.a

```
$ dagen -R type type | dag -Tps | $PRINT
```

Cycles in the picture usually represent linked lists. Anonymous structures or unions are numbered uniquely in the C program database.

For large programs, the directed graphs generated by *dag* may not fit on a single sheet of paper. *Dag* provides a pagination fature for partitioning the directed graphs if they are generated in the PostScript format. Some graph filters written in shell scripts are also provided for filtering out unnecessary details in various ways.

6 Doing More with the Program Database

The program database provides a foundation for building software tools that investigate program internals in various ways. This section briefly introduces several tools that perform some interesting functions: subsystem extraction, dead code elimination, and software restructuring. These tools are collectively called the Investigator.

6.1 Subsystem Extraction

In some projects, it might be necessary to extract one or more self-contained subsystems from an existing system for various reasons. For example, suppose the global variable declaration `ftable` in Figure 3 is to be extracted and reused in a different system called *TabOp*. Note that the variable declaration alone will not compile in the new system because there are missing references. Eventually, all objects referred to directly or indirectly by `ftable` need to be extracted and copied over to *TabOp*. The same problem arises in reusing functions written by other people.

Subsys is a tool that computes the *reachable set (subsystem)* of any object by tracing its relationships with other objects. For example, the reachable set of `ftable` consists of three macros (`FNUM, PFPC, DIM`), two data type definitions (`COOR, struct coor`), and two function definitions (`shift, rotate`). They must be brought over to *TabOp* in order for the declaration of `ftable` to compile.

The definitions of all objects in a reachable set can be bundled together automatically by invoking `viewdef` on each object and redirecting the output to a file. Subsystem extraction is also a useful mechanism for partitioning DAG pictures derived from large program databases.

6.2 Dead Code Elimination

Normally, if a global object (except `main`) is not referred to by any other objects, then it is a good candidate for deletion. For example, the function `fill_spec_table` shown in Figure 4 has a fan-in of zero and should probably be deleted.

Before deleting any object, we need to determine its *delete set*, which specifies all the dead code associated with it. The delete set of an object A, *D(A)*, consists of all the objects in its reachable set, *R(A)*, that do not belong to the reachable set of any object outside *R(A)*. For example, Figure 7 shows functions in the delete sets of the functions `fill_spec_table` and `rel.main`[7]. There are eight functions in the reachable set of `fill_spec_table`, but six of them are in the reachable set of other functions and therefore cannot be deleted. Note that the set of unshaded functions forms the reachable set of `example.main`, and this is not a coincidence.

[7] Although not shown here, all kinds of reference relationships defined in the CIA conceptual model should be considered in computing the delete sets.

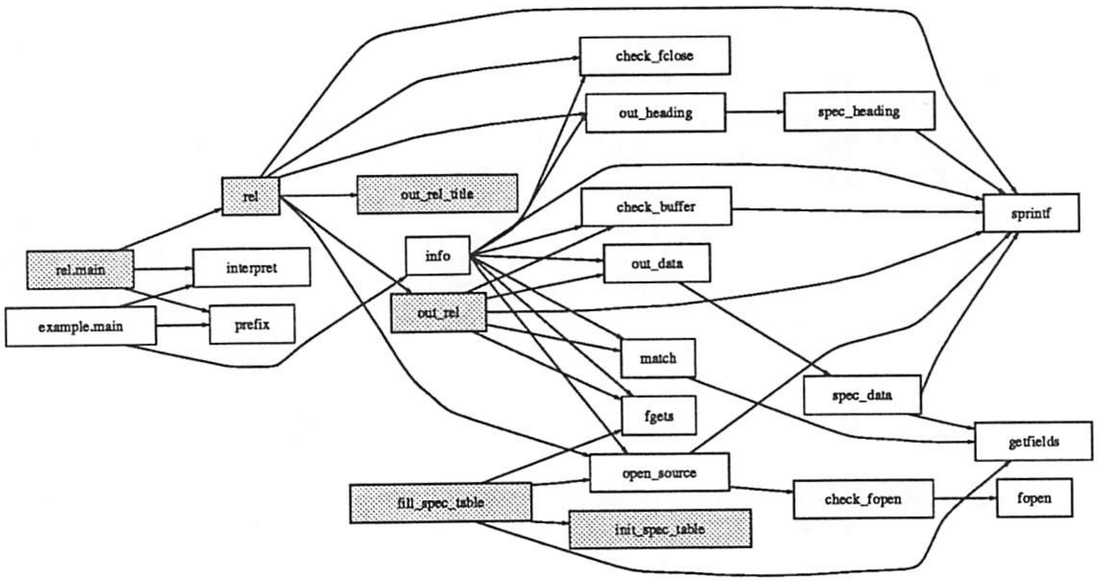


Figure 7: The Delete Sets of fill_spec_table and rel.main

6.3 Software Restructuring

Software maintenance activities usually cause program structures to deteriorate over time. When the entropy of a program reaches a certain point, it may be desirable to restructure it to save future maintenance cost.

For example, suppose it is desirable to move the static variable **ftable** shown in Figure 2 from **trans.c** to **op.c** because **ftable** is found to be more closely coupled with functions in the file **op.c**. Such a simple task may require a very detailed examination of the objects and relationships involved. For instance, the macro FNUM and the data type PFPC would have to be moved to **op.c** as well in order for the file to compile. The storage class of **ftable** has to be changed so that the function **trans** in **trans.c** can still refer to it. Also, the two external function declarations, **rotate()** and **shift()**, in **trans.c** should be removed. Finally, because both **trans.c** and **op.c** have to refer to PFPC, it is better to declare PFPC in **coor.h**.

A tool can use the program database to do this checking and then take proper actions automatically or provide suggestions to programmers. For a human programmer to do all of this checking in a large programming project is almost inconceivable. Many compilations will be required to correct the problems. For this reason, most project managers will not like the idea of restructuring dormant code even if the code contains obvious design deficiencies.

7 Building Large Program Databases

There are a few more things to consider when *cia* is used for a large programming project: incremental abstraction, preparation of makefiles, and the use of Nmake [6] rules.

7.1 Incremental Abstraction and Makefiles

The incremental abstraction facility in *cia* allows a program database to be rebuilt with a minimal effort after some changes are made. For each source file with a .c suffix, *cia* creates a .A file

metrics	*Ksh*	*Nmake*	*Gremlin*	*Cia*	*GNU Emacs*	*BSD Kernel*
number of files	110	35	165	42	109	395
source lines	26381	16480	21290	12066	117024	206398
source size (bytes)	523353	368725	574993	302864	3179230	5155320
number of symbols	2667	1118	3941	1120	4153	6451
number of references	4926	3899	4106	973	17977	19688
database entries	7593	5017	8047	2093	22130	26139
database size (bytes)	197057	130526	337486	58171	598068	642216

Table 2: A Comparison of Several Program Databases

to keep its relational information. *Cia* can link these `.A` files to create a program database. A programmer does not have to remember which files have been modified; a makefile is recommended for maintaining the program database. By constructing rules similar to those for C compilation, the *make* program can be used to maintain the program database easily.

7.2 Nmake Rules for CIA

The rule to build a C program database has been integrated in the latest *nmake* [6] rules. If *nmake* is used, then there is no need to modify your Makefiles for building the program database. For example, the *nmake* Makefile looks like the following for one of the CIA tools:

```
.SOURCE.h: /home/chen/include
.SOURCE.c: ../libpdb
DEBUG==1
map:: map.c exec.c field.c spec.c
```

To build a program database, simply type

```
$ nmake ciadb
```

Nmake scans source files to check for file dependencies, provides the proper `-I` and `-D` options for *cia* to run on each file, and then creates the program database by linking all `.A` files. Using *nmake* is particularly important for projects with complex source hierarchies. For example, many program databases reported in the next section were built using the viewpathing mechanism in *nmake* without modifying anything in their Makefiles or copying any source files.

7.3 Comparing Several Program Databases

Table 2 compares the source and database sizes of several programs. The source size of each program includes all the header files it uses, but excludes libraries shared with other programs. *Ksh* [12] is a shell programming language. *Nmake* is a program for maintaining and updating computer programs. *Gremlin* [13] is an interactive graphics editor running on SUN workstations[8]. Both the GNU Emacs and 4.3 BSD Kernel source are configured on a VAX 8650. The BSD Kernel source includes all device drivers and the TCP/IP code.

In most cases, the database size is less than half of the source size, but it depends on whether accelerator tables are used in building the database. Note that there are over 26,000 symbol declarations and references in the BSD kernel. Managing such information can be extremely difficult

[8]Figure 1 was drawn by using Gremlin.

without automated tools. Besides the programs shown in Table 2, program databases have been built for several propritary projects that are four to five times the size of the BSD kernel.

Cia normally takes less than 60 percent of the time it takes to compile a source file. The performance of InfoView commands is adequate even for a large program database. For example, on a VAX 8650 running 4.3 BSD, the following list shows the time (`user + sys`) it takes to execute some typical InfoView queries on the program database for the BSD kernel itself:

```
0.20 cpu seconds for "def function rwuio"
0.28 cpu seconds for "ref function - function rwuio"
0.23 cpu seconds for "viewdef function rwuio"
0.38 cpu seconds for "viewref function - function rwuio"
```

8 Current Status

The initial conceptual model of CIA was designed at University of California, Berkeley [5] . It consists of the five object kinds and the first four reference relationships shown in Table 1. The prototype implementation was partially sponsored by AT&T Bell Laboratories. To satisfy the needs of many large projects in AT&T, the CIA conceptual model was redesigned and expanded. The current system is highly portable and has been tested on variants of UNIX systems. In addition, InfoView was ported to the Macintosh by Charles Hayden.

Several graphical software browsers that use the C program database have been built. These browsers run on bit-mapped displays and effectively turn C program text into Hypertext [14] .

Besides the research work reported in Section 6, we are conducting experiments on software metrics and program structure comparison using information stored in the program database. We are also developing conceptual models for other languages and documents, including C++ [15] and Makefiles.

9 Acknowledgements

CIA makes extensive use of *libx.a* and *libpp.a* developed mainly by Glenn Fowler. Charles Hayden designed an indexed database schema and rewrote InfoView to improve its functionality and query performance. Michael Nishimoto, Lenora Eng, Wen-Ling Chen, Benjamin Chang, Scott Nishimoto, and Andy Hung all contributed to the early prototype tools of CIA. In addition to the aforementioned people, many Bell Labs colleagues also made suggestions that affect design decisions in the latest version of CIA.

Glenn Fowler and David Korn provided a *shipin/shipout* facility for the distribution of CIA. Terry Anderson provided the latest TM macros for formatting this document. David Belanger, Glenn Fowler, Judith Grass, Jon Helfman, Eduardo Krell, Bob Parnass, David Rosenblum, Valerie Torres, Kiem-Phong Vo, and Mike Wish reviewed a draft of this paper and provided comments that improved its quality. Finally, I would like to thank Professor C. V. Ramamoorthy at the University of California at Berkeley, the GRIST group at Bell Labs in Columbus, and David Belanger for the encouragement I needed during early development stages of CIA.

References

[1] Warren Teitelman and Larry Masinter. The Interlisp Programming Environment. *IEEE Computer*, 14(4):25–34, April 1981.

[2] Joseph. L. Steffen. Interactive Examination of a C Program with Cscope. In *USENIX Association Winter Conference Proceedings*, pages 170–175, January 1985.

[3] M. A. Linton. Implementing Relational Views of Programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1984.

[4] M. Stonebraker, E. Wong, R. Kreps, and G. Held. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[5] Yih-Farn Chen and C. V. Ramamoorthy. The C Information Abstractor. In *The Tenth International Computer Software and Applications Conference (COMPSAC)*, pages 291–298, October 1986.

[6] G. S. Fowler. The Fourth Generation Make. In *USENIX Portland 1985 Summer Conference Proceedings*, pages 159–174, 1985.

[7] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. AddisonWesley Publishing Co., 1988.

[8] E. R. Gansner, S. C. North, and K. P. Vo. DAG – A Program that Draws Directed Graphs. *Software: Practice and Experience*, 18(11), November 1988.

[9] American National Standards Institute. Draft Proposed American National Standard for Information Systems – Programming Language C, May 1988. Doc. No. X3J11/88-090.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (second edition)*. Prentice Hall, 1988.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[12] Morris I. Bolsky and David G Korn. *The KornShell – Command and Programming Language*. Prentice Hall, 1988.

[13] Mark Opperman, Jim Thompson, and Yih-Farn Chen. A Gremlin Tutorial for the SUN Workstation. Technical Report UCB/CSD 322, Computer Science Division, University of California, Berkeley, December 1986.

[14] Jeff Conklin. Hypertext: An Introduction and Survey. *IEEE Computer*, 20(9):17–41, September 1987.

[15] Bjarne Stroustrup. *The C++ Programming Language*. AddisonWesley Publishing Co., 1987.

An Efficient File Hierarchy Walker

Glenn S. Fowler
David G. Korn
K.-Phong Vo

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper presents an interface specification and an efficient implementation of a general purpose library routine, *ftwalk*, to traverse a UNIX* file system hierarchy. A number of standard file system utilities, e.g., **find**, **ls**, **rm**, and others have been reimplemented using *ftwalk*. The total source code size is 30% smaller and the efficiency of all commands improves. More importantly, these commands now handle the file system search in a uniform, robust, and secure manner.

New tools have been built with *ftwalk*. A file system perusing tool, **tw**, will be described. **tw** subsumes the functionality of **find** and **xargs**. Further, it provides a powerful expression language with a syntax similar to C. For typical applications in which commands are executed on generated file names, **tw** is 5 to 10 times faster than **find**. The combination of a powerful language and performance efficiency in **tw** should reduce the practice of adding directory recursion to commands.

1. Introduction

Many standard file system utilities such as **ls**, **rm**, **find** and others either require or provide options to traverse file hierarchies exhaustively. Given that traversing a file hierarchy is a common operation, it is desirable to have a standard subroutine implementation for commands that require this functionality. There have been attempts at providing such a routine[ftw][Lin]. In particular, System V and 9th Edition C libraries provide different implementations of a routine, *ftw*, designed for this purpose. Surprisingly, no attempt has been made to use the routine in implementing the standard commands (except for 9th Edition **du**). As a result, each command has its own file system walker with a separate set of quirks and problems. As examples, the 4.3BSD **ls** command cycles forever if there is a loop in the file hierarchy caused by symbolic links, the **rm** command cannot remove a file hierarchy a few hundred levels deep, and the **find** command cannot search a file hierarchy deeper than the number of available file descriptors. Inertia is a likely cause for not using a common file system walker in standard commands. However, on closer inspection, we also found that current implementations of *ftw* are either too inefficient for general use or simply inadequate in the interface to support the variety of requirements from different commands. The interface inadequacy is further stressed when we want to build commands that are portable across different UNIX flavors such as System V and 4.3BSD that have wildly different file system structures. For these reasons, we decided to design and implement a new file system walking routine. It seems reasonable that such a routine should exhibit the following qualities:

- *Generality*: The routine must be general enough to support all current standard file system utilities and other tools that we can think of.
- *Efficiency*: The routine must perform about as well as its hand-coded counterparts in current commands.

* UNIX is a registered trademark of AT&T.

- *Finiteness*: The routine should terminate even in the event of cycles caused by directory links and/or symbolic links. Such cycles must be detected and avoided.
- *Robustness*: The routine should work regardless of the depth and extent of the search. For example, the depth of a file hierarchy should not in itself be a limiting factor. John Linderman pointed out to us that this is an important security consideration in preventing hostile users from creating deep files that may be hidden from current file system utilities.
- *Portability*: The routine should work on all systems that we have access to, e.g., System V, BSD, HP/UX, 9th Edition. In addition, it should be conformant to POSIX 1003.1[IEEE].

We have designed and built a file walking routine *ftwalk* that satisfies the above quality requirements. The extensive interface features of *ftwalk* were carefully designed so that all standard utilities (**cp**, **du**, **find**, **ls** and **rm**) could be rebuilt using *ftwalk*. We have, in fact, rebuilt these commands. Inheriting the same qualities in *ftwalk*, the new commands handle file system traversal in a consistent and robust manner. In addition, their efficiency is at least as good or much better than their earlier counterparts. The generality of *ftwalk* also enables the construction of new tools. An example is **tw**, a new command that both generalizes and subsumes the functionality of **find** and **xargs**. A novel feature of **tw** is a powerful expression language with a C-like syntax. For recursive execution of commands on a file hierarchy, **tw** is typically between 5 to 10 times faster than **find**. Comparing with commands such as **chmod** and **chgrp**, **tw** comes to within 20% in system time and about the same on CPU time.

The rest of the paper is organized as follows. Section 2 describes *ftwalk*. Section 3 discusses issues and problems in reimplementing standard utilities. Section 4 describes **tw**. Finally, in the conclusion, we indicate some future directions.

2. The *ftwalk* Routine

2.1 Interface Specification

The interface of *ftwalk* contains many options designed for two purposes: *generality* and *efficiency*. Generality means that the interface must provide enough abstractions to support all commands that we know of or want to build. Often, a general abstraction can prevent efficient implementations of particular commands. In such cases, there are features allowing an application to "control" an abstraction to achieve the desired level of efficiency. For example, by default, *ftwalk* gets file status of all objects as soon as they are found. This enables the walker to determine terminal objects such as files or unreadable directories and process them immediately. However, getting status information of objects is an expensive operation. Thus, an option `FTW_DELAY` is provided to control this feature for a command such as **ls** that may not need such information for directory children (e.g., in a plain **ls** call).

Figure-1 shows the function prototypes of *ftwalk* and the user-supplied functions *userf* and *comparf* and the basic data structure `struct FTW`. In a call to *ftwalk*, a depth-first search starts at the directory `path`. Though there are many different graph search techniques[AHU], depth-first search is the common search method in all the commands that we studied. In the case of commands such as **rm** and **du**, a postorder depth-first search is, in fact, the method of choice. This is because the descendants of a directory must be processed before the directory itself can be processed.

The user-supplied function, *userf*, is called at visits of objects. It has a single argument, a pointer to a `struct FTW` structure that stores information on the object being visited. *userf* is called exactly once on a terminal object such as a file or a directory causing cycles, For a normal directory, *userf* may be called once or twice in preorder and/or postorder. As discussed below, it is possible to prune a search after a preorder visit or to restart a search after a postorder visit by setting appropriate values in the `ftw->status` field in *userf*. *userf* should normally return `0`. A non-zero return value from *userf* causes *ftwalk* to return immediately with the same value.

The function, *comparf*, if supplied, is used to order the original paths (see `FTW_MULTIPLE` below) and to order the children of other encountered directories. Note that such an ordering directs the search.

```
int ftwalk(char *path, int (*userf)(), int flags, int (*comparf)());
int userf(struct FTW *ftw);
int comparf(struct FTW *ftw1, struct FTW *ftw2);

struct FTW
{
        struct FTW          *link;
        struct FTW          *parent;
        void                *local;
        struct stat         *statb;
        char                *path;
        short               pathlen;
        unsigned short      info;
        unsigned short      status;
        short               level;
        short               namelen;
        char                name[];
};
```

Figure 1. *Ftwalk* and related functions and data structures

This is important for commands such as **ls** that need to traverse a file hierarchy in some specific order. *Comparf* is called with two arguments, `ftw1` and `ftw2`, that point to the `struct FTW` structures of the objects being compared. *Comparf* should return a negative, zero, or positive integer to indicate whether `ftw1` is considered smaller, equal or larger than `ftw2`.

The argument `flags` of *ftwalk* is a bit vector. Currently, the following bits are supported:

`FTW_DOT`: *ftwalk* will not attempt to change directory (*chdir*) while recursively searching directories. Thus, throughout the search, the current directory (dot) stays the same. `FTW_DOT` should not be used often since changing directory during a search is a major efficiency gain in system time. `FTW_DOT` is needed for applications that try to avoid undesirable file system side effects (e.g., core dumps) in unexpected places. Generally, changing directory is only an option, not a requirement. A successful search depends only on appropriate read permission, not search permission.

`FTW_CHILDREN`: Preorder visits of directories are implied even if `FTW_POST` (below) is turned on. In such a visit, the `ftw->link` field contains a linked list of all children objects of the directory being visited. This option is useful for commands (e.g., **ls**) that need the children list for local processing purposes (e.g., computing multi-column format).

`FTW_DELAY`: When `FTW_CHILDREN` is turned on, the fields `ftw->statb (struct stat)` of children objects remain undefined until these objects are visited. This helps a command such as **ls** to avoid expensive computations when only the names of the children of a directory are needed.

`FTW_MULTIPLE`: The argument `path` of *ftwalk* is really of type `char**` and points to a null-terminated array of path names. *Ftwalk* will search all hierarchies rooted at these paths.

`FTW_PHYSICAL`: Symbolic links will not automatically be followed. Each symbolic link object will initially be treated as a terminal object. The user function *userf* may ask *ftwalk* to follow the link by setting the field `ftw->status` to `FTW_FOLLOW` (below).

`FTW_POST`: For non-terminal directories, *userf* is called only on postorder visits. Note that if `FTW_CHILDREN` is turned on, preorder calls to *userf* will always be performed.

`FTW_TWICE`: For non-terminal directories, *userf* is called twice in both preorder and postorder.

The use of some of the fields in the structure `struct FTW` have been mentioned. Below are detailed descriptions of these fields.

`link`: This field is used in two different ways. For a directory that causes cycles, `link` points to the object on the current search path that is identical to this directory. For any other directory, in an `FTW_CHILDREN` preorder visit, `link` points to the list of children of the directory.

`parent`: This field points to the parent directory of the object being visited. Thus, ancestors of an object can be traced via the `parent` pointers. For convenience, a parent structure is also provided for level 0 objects. It contains the same file status information as that of its child.

`local`: This field is provided to users to store any information local to the object being visited. It is initialized to `NULL` by *ftwalk*. Note that `local` is wide enough to store pointers. It is the user's responsibility to free heap allocated data stored in the `local` fields. This can be done, for example, in a postorder visit.

`statb`: This field stores the `struct stat` buffer containing all file system information available on the object. Note that if the option `FTW_DELAY` is turned on, such information is undefined when the object is on the children list of a directory being visited in preorder (`FTW_CHILDREN`).

`path` and `pathlen`: These fields contain the path name and the path name length of the object being visited. The `path` buffer is guaranteed to be large enough so that the name of any child object of the current object can be appended.

`info`: The type of the object being visited. Following are the types:

`FTW_NS`:	The type of this object is unknown because *stat* or *lstat* failed.
`FTW_F`:	This object is a file.
`FTW_SL`:	This object is a symbolic link.
`FTW_D`:	This object is a directory being visited in preorder.
`FTW_DC`:	This object is a directory that causes cycles.
`FTW_DNR`:	This object is a directory that is not readable.
`FTW_DNX`:	This object is a directory that is readable but not searchable.
`FTW_DP`:	This object is a directory that is being visited in postorder.

`status`: This field is used in two different ways. In the call to *userf*, it is set to either `FTW_NAME` or `FTW_PATH` to indicate whether the user function should use the base `name` or the `path` name to get file system status of the object. This is important for correct processing when the option `FTW_DOT` is off. If `FTW_DOT` is on, the value of `status` is always `FTW_PATH`. In return, `userf` can set `status` to one of the below values:

`FTW_AGAIN`: If this is a postorder visit, descend the hierarchy rooted at this object again.

`FTW_NOPOST`: This is usually set in a preorder visit of an object. It suppresses any postorder visit to this object.

`FTW_FOLLOW`: If this is a symbolic link, follow it.

`FTW_SKIP`: Prune the search at this object. Thus, descendants of this object will not be visited.

`FTW_STAT`: This value can be assigned to the `status` fields of children of the current object when `FTW_DELAY` and `FTW_CHILDREN` were turned on. It means that the `statb` structure of the respective child has been filled by *userf*.

`level`: This field contains the depth from the root object of the object being visited.

`name` and `namelen`: These fields contain the base name and name length of the current object.

2.2 The *ftwalk* Search Algorithm

ftwalk employs a non-recursive depth-first search to walk file hierarchies. Using a non-recursive version of depth-first search allows us to handle various exceptions in a clean manner. In a recursive implementation, an exception may necessitate the unraveling of several recursion layers with extra book-keeping. The non-recursive algorithm also simplifies optimizations such as reducing the number of back-up *chdir* calls by eliminating the need to use static variables to retain states across recursive calls.

```
 0.    struct FTW *todo, *ftw, *first, *last, *f;
 1.    todo = top level path;
 2.    while(todo)
 3.    {
 4.            ftw = todo;
 5.            if(preorder processing is needed for ftw)
 6.                    userf(ftw);
 7.            if(ftw is a directory that does not cause cycles)
 8.            {
 9.                    first = last = NULL;
10.                    for(each child f of ftw)
11.                    {
12.                            if(first == NULL)
13.                                    first = last = f;
14.                            else    last = last->link = f;
15.                    }
16.                    if(last)
17.                    {
18.                            last->link = todo;
19.                            todo = first;
20.                    }
21.            }
22.            while(todo && todo == ftw)
23.            {
24.                    if(postorder processing is needed for ftw)
25.                            userf(ftw);
26.                    ftw = ftw->parent;
27.                    todo = todo->link;
28.            }
29.    }
```

Figure 2. A non-recursive file system depth-first search

Ignoring detailed interface features such as turning off FTW_DOT and sorting objects, Figure-2 shows a skeleton of the non-recursive depth-first search algorithm in a quasi-C syntax. Note that internal to the algorithm, the `link` field of `struct FTW` is used to maintain the list of elements being searched. The following discussion is based on the line numbers in Figure-2.

`0-1`: These statements declare relevant variables and initialize the `todo` list with the root path of the hierarchy to be searched.

`2-29`: This is the main loop of the algorithm. The variable `todo` always points to the element to be processed next. The `todo` list is basically a stack in which the last element put on it is the next element to be processed.

`4-6`: The variable `ftw` is set to the element to be processed. If a preorder processing of `ftw` is required, it is done unless `ftw` is a directory and the option `FTW_CHILDREN` is turned on. In that case, the code on lines 5 and 6 for preorder processing is actually done after line 18 following the construction of the children list. If `FTW_DOT` is off, the current directory is set to `ftw` before its children are read. This speeds up various system calls such as *stat* or *open*.

`9-20`: The list of children of `ftw` is constructed. Since all children of a directory are read, the algorithm only needs one open file pointer at any given time. For efficiency, if *comparf* is given, the children of a directory are sorted using a stable insertion sort based on a self-adjusting binary tree[ST]. A similar tree is also used for efficient checking of cycles.

`22-28`: The stack is popped. This corresponds to subroutine returns in a recursive implementation of depth-first search. If post-order processing of an element being popped from the stack is required, the user function *userf* is called. Though not shown in the code, The number of popped levels is also recorded to optimize the number of *chdir* calls to back up the current directory.

2.3 An Example

In the appendix, we show the code of a user function, *draw*, to generate the description of a file hierarchy in the DAG language[GNV]. Figure-3 shows a generated DAG drawing of a small hierarchy. In the drawing, boxes indicate directories, ellipses files, and diamonds symbolic links. Dotted edges represent either symbolic links or hard links.

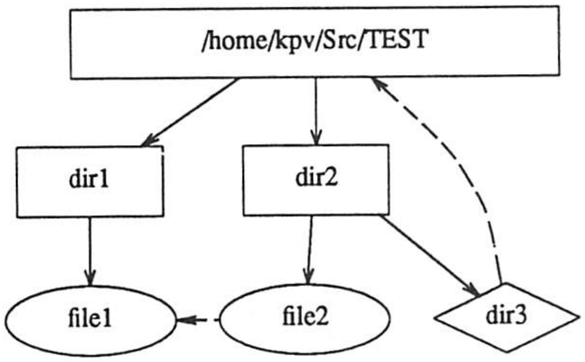


Figure 3. A file hierarchy

Following is the code fragment to invoke *ftwalk* to walk and draw a given file hierarchy:

```
printf(".GS\n");
ftwalk(path,draw,FTW_PHYSICAL,NULL);
printf(".GE\n");
```

The two `printf` statements generate the prologue and epilogue of a DAG description. The root of the hierarchy to be drawn is given in `path`. `FTW_PHYSICAL` is turned on to detect symbolic links. Since the relative order of directory entries is not important, `comparf` is set to `NULL`.

3. Reimplemented Standard Commands

A good way to test the applicability of *ftwalk* is to reimplement standard utilities that peruse file systems. This allows performance, feature and coding style comparisons. Further motivating factors are to enforce consistency and to enhance robustness across these commands:

Logical vs. Physical: A logical search automatically follows symbolic links to their physical counterparts while a physical search stops at symbolic links. With *ftwalk*, both types of search can be provided as command options without too much programming effort.

Cycle Detection: Cycles caused by both symbolic and hard links on directories are automatically detected. This allows commands to default to the logical search mode without fear of non-termination, a major problem on systems with symbolic links.

Robustness: A good deal of effort went into *ftwalk* to remove artificial constraints. As an example, *ftwalk* can handle directory structures of arbitrary depths (up to virtual or hard memory limits). Current implementations of important standard utilities such as **find** and **rm**, in fact, fail on deep directory structures.

An informal survey of BSD, System V and 9th Edition machines produced the following commands that do some form of file hierarchy traversal: **chgrp**, **chmod**, **cp**, **diff**, **du**, **find**, **ls**, **rm**, and **tar**. Instead of blindly doing wholesale changes to these commands, we study their interfaces and the interrelationships in their use. This allows us to reimplement and extend certain commands selectively while leaving others untouched. We discuss below the issues and problems in rebuilding these commands.

chgrp and **chmod**: The **−R** (recursive) option added to the BSD versions of **chgrp** and **chmod** can be simulated by using these commands in conjunction with **find** as in the following shell[BK] statements:

```
find dir -exec chmod permissions "{}" ";"
chmod permissions $(find dir -print)
```

The first statement always works but it is slow, since a separate **chmod** process is created for each generated file name. The second command is faster but it does not always work since it may exceed the argument limit to the system call *exec*. To avoid the temptation of adding `-R` to all our favorite commands (``grep -R pattern .'' might be nice), we changed the `-exec` primitive of **find** to accept + (plus) as a command terminator in addition to the usual ; (semicolon). The + instructs **find** to execute the command a minimal number of times in the manner of **xargs** (see also **tw** below).

cp: The `-r` option allows copying directory hierarchies (9th Edition, provides **reccp** to accomplish the same thing). This operation is better accomplished by the POSIX ``**pax −rw**'' command[IEEE] (similar to ``**cpio −p**'') since **pax** provides better control over the attributes of the copied files. With this in mind, **cp** was originally redone to execute ``**pax −rw**'' for the recursive case. However, a recoding of **cp** using *ftwalk* is more efficient even in the non-recursive case by taking advantage of the information provided in the `struct FTW` structure of *ftwalk*.

diff: This command has an option to compare two directory hierarchies. This exposes a limitation in a function interface such as *ftwalk* that cannot allow simultaneous explorations of two or more different hierarchies. We shall come back to this issue in the conclusion.

du: This command is simplest of all the commands to reimplement. In reimplementing it, we found the need to have a place to store summaries of information of descendants of directories on a search path. The `ftw.local` field of `struct FTW` fulfills this need. The simplicity of **du** is one of the guiding factors in the design of the **tw** expression evaluator described below.

find: This is an indispensible command but it commits a terrible sin of robustness: it will not search beyond a depth higher than the number of available file descriptors (about 20 for our system). This means that files in relatively shallow hierarchies may never be found. Further, the expression interface of **find** is messy, making any extension difficult. The new command `tw` described in Section 4 corrects these problems.

ls: The column output mode (`-C` option) of this command requires that all children of a directory are available when it is visited in preorder. The option `FTW_CHILDREN` of *ftwalk* solves this problem. In addition, providing a sort function, *comparf*, to *ftwalk* efficiently solves the problem of sorting directory entries.

rm: This command varies the most among different implementations, none of which deal with hard link directories properly. Although hard links to directories are rare, a standard command in `/bin` should be able to handle them. There are two different system calls to remove file entries, *rmdir* and *unlink*. Roughly, *unlink* simply removes an entry while *rmdir* removes a directory entry and the implied "hard links" `.` (dot) and `..` (dot-dot). In the presence of hard links, care must be taken as to which of these system call to use. Our reimplemented **rm** solves this problem. Using *ftwalk* also solves a robustness problem with previous implementations where deep hierarchies cannot be removed.

tar: On our system, **tar** is replaced by a local implementation of the POSIX command **pax**, which uses *ftwalk* for file system traversal.

Table-1 and Table-2 summarize comparative performance statistics for the commands: **cp, du, find, ls** and **rm.** Table-1 gives statistics for the old commands while Table-2 gives statistics for the reimplemented commands. The statistics include numbers of non-commented source lines (NCSL), and CPU times, System times, and their total. The times are measured in seconds and result from running the programs on `/usr/src` on our local system which is a VAX 8650 running 4.3BSD. This directory system has 201 directories and a total of 3480 entries. To reduce statistical fluctuations, each timing result is an average of three runs at night when the machine is largely idle.

Program	NCSL	CPU	System	CPU+Sys
cp	211	1.72	37.06	38.78
du	152	.47	3.50	3.97
find	1053	.29	3.65	3.94
ls	616	1.60	5.29	6.89
rm	218	.91	17.90	18.81
Total	2250	4.99	67.40	72.39

TABLE 1. Statistics of old commands

Program	NCSL	CPU	System	CPU+Sys
cp	208	1.02	36.27	37.29
du	124	.33	3.46	3.79
find	676	.48	3.44	3.92
ls	499	.75	3.80	4.55
rm	184	.62	12.28	12.90
Total	1691	3.20	59.25	62.45

TABLE 2. Statistics of reimplemented commands

As Table-1 and Table-2 show, overall, there is a reduction of about 30% in source code size. Not all of the code reduction is due to using *ftwalk* since the commands have been rewritten extensively. The combined time performance for all commands improves about 15%. Major performance gains are in **ls** and **rm**. Both of these commands benefit from *ftwalk*'s efficient utilization of system calls. The more than 100% improvement in CPU time for **ls** is due to *ftwalk*'s use of an efficient search tree for sorting directory children.

4. The tw Command

The **find** command is a popular tool for recursive processing of objects in file hierarchies. However, find has several drawbacks that limit its use. The most visible problems are **find**'s unconventional expression syntax and its use of shell meta-characters as arguments which necessitates awkward quoting. Worse than **find**'s awkward interface are useful options whose semantics cause extreme inefficiency. For example, the `-exec` option requires a separate invocation of the specified program for each generated file name.

To alleviate the efficiency problem in **find**, System V provides **xargs**, a command which reads file names, one per line, and applies a given command to as many files at one time as possible. As an example, below are two shell command lines that perform more or less the same function though the second one is more efficient on a large directory structure.

```
find path -exec program "{}" ";"
find path -print | xargs program
```

A problem with using **xargs** in this manner is that file names that contain the `\n` (new-line) character cannot be handled. This is a potential security problem as the operating system does not preclude the creation of such files.

Aside from **find** and **xargs**, a popular approach in BSD is to add a recursive option to individual commands. This requires modification of every command of interest and lacks the useful pattern matching mechanism of **find** for selective command execution.

Inadequacies in current approaches expose the need for a new language and tool to interface to the file system structure that is: flexible to use, efficient, and easily extensible to match frequent changes in file system interface structures. In this spirit, we created a new command, **tw**, whose interface language is C-like and provides extensive access to file information. Like **find**, **tw** recursively traverses a file hierarchy and performs actions on visited files. Like **xargs**, **tw** only executes a given command a minimal number of times. The rest of this section describes **tw** and gives examples of how to use it.

4.1 Interface Description

Without arguments, **tw** is equivalent to "`find . -print`". When given a command, **tw** executes the command with the generated file names as arguments. The given command may be executed more than once depending on the limit of the *exec* system call. Similar to **xargs**, **tw** can also read file names from the standard input if its first argument is given as `-`. As an example, below are four commands that performs equivalent functions:

```
tw chmod o-w
chmod -R o-w
find . -print | xargs chmod o-w
find . -print | tw - chmod o-w
```

Conditional C-style expressions may be specified with the `-e` option. The expression operands are the fields of the `struct FTW` and `struct stat` structures (the prefix "`st_`" is deleted) of the object being visited. Here is a command to list files with *group* or *other* write permissions:

```
tw -e "mode & 022"
```

Some fields allow `"..."` and `'...'` string operands. The first command below is equivalent to the above command but somewhat more readable. The second command lists all objects modified since yesterday morning (most common time and date expressions are allowed).

```
tw -e "mode & 'go+w'"
tw -e "mtime >= 'yesterday 8am'"
```

The `'...'` strings are handy when combined with `"` shell quoting. Strings may be compared, where `==` and `!=` treat the right hand operand as a **ksh**[BK] file match pattern. The following command executes `grep pattern` on a selected collection of source files:

```
tw -e "name == '*.[chly]'" grep pattern
```

For convenience, the type bits of the field `st_mode` in `struct stat` are placed in a separate field. For example, the following lists all subdirectories of the current directory:

```
tw -e "type == DIR"
```

Identifiers may be prefixed by one or more "`parent.`" to reference parent directory status information. The below command prunes all directories that are in a different file system type (e.g., *nfs* vs. *rfs*) than their parent directories:

```
tw -e "if (fstype != parent.fstype) status = SKIP"
```

Explicit file references can also be done by prefixing an identifier with '`filename`'. Following is an example script that lists all files newer than the file `/etc/backup.time`.

```
tw -e "mtime > '/etc/backup.time'.mtime"
```

The complete list of predefined identifiers appears below. Most correspond to the `st_` elements of `struct stat`. New fields can be added as `struct stat` grows.

`atime`: Most recent access time. If `atime`, `ctime`, or `mtime` is the left hand side operand of a binary operator, the right hand side operand may be a string interpreted as a date expression.

`blocks`: The number of blocks in the file.

`ctime`: The inode change time.

`dev`: The device number.

`fstype`: The file system type name. The default value is `ufs`.

`gid`: The group id number. If `gid` is the left hand side operand of a binary operator, the right hand side operand may be a string that is interpreted as a group name.

`gidok`: `1` if `gid` is a valid group id in the system database, 0 otherwise.

`ino`: The inode or file serial number.

`level`: The depth of the file relative to the starting directory.

`local`: The `local` field of `struct FTW`. This field may be assigned certain values.

`mode`: The identifier `FMT` is recognized and can be used to mask the file format and permission bits. If `mode` is the left hand side operand of a binary operator, the right hand side operand may be a string that is interpreted as a **chmod** permission expression.

`mtime`: The time at which the object was last modified.

`name`: The file name of the object with no directory prefix.

`nlink`: The number of hard links.

`path`: The full pathname of the current object. Note that this is only defined for the current object. For example, `parent.path` is undefined.

`perm`: The file permission bits of `mode`. If `perm` is the left hand side operand of a binary operator, the right hand side operand may be a string that is interpreted as a **chmod** permission expression.

`rdev`: The device numbers for special files.

`size`: The number of bytes in the file.

`status`: The `status` field of `struct FTW`. This field may be assigned one of: `AGAIN`, `FOLLOW`, `NOPOST`, and `SKIP` which correspond to the *ftwalk* constants prefixed by "`FTW_`". For example, `status=SKIP` may be used to prune subdirectories from the search.

`type`: The file type bits of `mode`. The identifiers `BLK`, `CHR`, `DIR`, `FIFO`, `LNK`, `REG`, and `SOCK` are recognized.

`uid`: The user id number. If `uid` is the left hand side operand of a binary operator, the right hand side operand may be a string that is interpreted as a user name.

`uidok`: `1` if `uid` is a valid user id in the system database, `0` otherwise.

`visit`: A variable (initialized to `0`) that can be updated on each visit to the file. A file is uniquely identified by its `st_dev` and `st_ino` numbers.

Expressions in **tw** are labeled to indicate different types of processing. The default label for an expression is `select`. Possible labels are:

`action`: This expression defines the action to be executed on selected file names (below). The default action is to list the selected file names on the standard output.

`begin`: If defined, this expression is evaluated once before the file system traversal starts.

`end`: If defined, this expression is evaluated once after the file system traversal ends.

`select`: This expression is used to select files to be acted upon. If it evaluates to true (non-zero) then the `action` expression is evaluated. The default value for `select` is `1`.

`sort`: This expression specifies an identifier by which child entries are sorted. The below example lists files sorted by newest to oldest `mtime` (the `!` inverts the sort sense).

```
tw -e "sort: !mtime"
```

Finally, `if-else`, `printf` and variable declarations round out the expression language.

4.2 Examples and Discussions

Figure-4 shows a **tw** script to list the name and inode number of each file and to list the total number of files encountered. Note that the variable `count` is automatically initialized to `0`.

```
tw -e "
        int count;
action:
        count++;
        printf('name=%s inode=%08ld\n', name, ino);
end:
        printf('%d file%s\n', count, count == 1 ? '' : 's');
"
```

Figure 4. A **tw** script to list file names and inode numbers

Figure-5 shows a script to emulate the **du** command. Here, the `-p` option specifies a postorder traversal and the `-P` option specifies a physical (don't follow symbolic links) traversal. Note how the `local` field is used to accumulate the total number of blocks contained in a directory subtree and the `visit` field check is used to ensure that hard links are counted only once. This script is about 40% slower than the standard **du** command, where 33% of the difference is in CPU time. This is reasonable since the action expression must be evaluated interpretively for each file in the search.

```
tw -pP -e "
action:
        if (visit++ == 0)
        {
              parent.local += local + blocks;
              if (type == DIR)
                      printf('%d\t%s\n', local + blocks, path);
        }
"
```

Figure 5. A **tw** script to emulate **du**

To measure system time overhead caused by the *fork* and *exec* system calls in **tw** and combinations of **find** and **xargs**, we ran tests on /usr/src, a large hierarchy, using the below three commands:

```
tw -d /usr/src null
find /usr/src -exec null ";"
find /usr/src -print | xargs null
```

Here, **null** is a command that does nothing. Using **null** instead of another standard command such as **wc** or **ls** allows us to measure only the system time overhead caused by *fork* and *exec* and not the cost of "real" work being performed by these commands. Table-3 shows the comparative timing results. The **tw** version is about 10 times faster than the **find** version and about 6.5 times faster than the combination of **find** and **xargs**. The relatively bad timing result for the **find** and **xargs** combination is largely due to the poor implementation of **xargs**.

Program	CPU	System	CPU+Sys
tw	.55	5.45	6.00
find	1.46	55.81	57.27
find | xargs	5.39	34.44	39.83

TABLE 3. Statistics for **tw**, **find** and **xargs**

5. Conclusions

We presented a new file hierarchy traversal routine, *ftwalk*, and its use in reimplementing certain standard file system commands and in implementing new commands. The applicability of *ftwalk* in a wide variety of programs shows the generality of its interface. The total source code size for the reimplemented commands reduces by 30% despite the addition of new options in some cases. Timing results show that these commands perform at least as well as before. In cases such as **ls** or **rm**, performance improves significantly. More importantly, all commands handle the file hierarchy traversal in a consistent and robust manner. Robustness is crucial for security concerns. Certain important commands such as **find** and **rm** previously could not handle deep hierarchies. We have tested our versions of these commands on directory structures deeper than 2000 levels.

New commands such as **pax**, a new POSIX conformant file archiver, and **tw**, a file system walker, have been built based on *ftwalk*. The latter command, **tw**, was described here. It subsumes the functionality of **find** and **xargs**. In addition, **tw** provides a powerful expression language with a syntax that resembles the C language. The expression language for **tw** is designed to be easily extendible to match any future changes in the file system interface structures. The performance of **tw** is many times better than **find** when a given command must be executed against generated file names. For a command such as **chmod (chgrp)**, `tw chmod` performs nearly as well as its hand-coded counterpart `chmod -R`. The `tw chmod` version is only about 20% slower in system time than the `chmod -R` version while CPU time is the same. The efficiency level and the availablility of a powerful language in **tw** should stop the practice of adding a recursive option to every command in sight.

Finally, one way of looking at *ftwalk* is that it outputs an ordered stream of objects from a file hierarchy. The order is defined by local orderings of children of directories and by the depth-first search. An interesting observation that arises from studying the non-recursive depth-first search algorithm of *ftwalk* is that the algorithmic steps are encoded in the `todo` data structure. This opens the possibility of implementing an interface along the line of *opendir* and *readdir* in which hierarchies are opened and read. The order in which an object is read will be exactly the same as the order that *ftwalk* would produce if it was called on the root of the respective hierarchy. This neatly solves a major problem for programs such as **diff** that need to manipulate multiple hierarchies simultaneously.

References

[AHU] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Algorithms*, Addison-Wesley, 1974.

[GNV] E.R. Gansner, S.C. North, and K.P. Vo, *DAG, A Program to Draw Directed Graphs*, Soft. Prac. and Exp., 1988.

[BK] M. Bolsky and D.G. Korn, *The KornShell Command and Programming Language*, Prentice Hall, 1988.

[ftw] *ftw*, System V Programmer's Reference Manual and 9th Edition UNIX Time-Sharing System Programmer's Manual.

[IEEE] *The IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE, 1988.

[Lin] J.P. Linderman, *dwalk*, Private communication, 1988.

[ST] D. Sleator and R.E. Tarjan, *Self-Adjusting Binary Trees*, Proc. 15th Ann. ACM Symp. on Found. of Comp., 1985.

Appendix

Below is a user function `draw` that can be used with *ftwalk* to draw file hierarchies.

```
#include      <ftw.h>

draw(ftw)
struct FTW    *ftw;
{
        int             linklev; /* level of linked object */
        char            *shape,  /* shape of this object */
                        name[256],/* unique name of this object */
                        link[256],/* name of its linked object if any */
                        parent[256];/* name of its parent */
        static int      N = 0;   /* to make unique name */

        if(ftw->info == FTW_SL)
        {      /* following symlink, set local so we know it was a symlink */
               ftw->status = FTW_FOLLOW;
               ftw->local = (VOID*) 1;
               return 0;
        }
        else if(ftw->info == FTW_NS)
        {      /* an object with no status */
               linklev = -1;
               sprintf(name,"ns%d",++N);
        }
        else
        {      /* linkobj gets the level of the object linked to ftw.
                  If there is no such object, it returns -1. */
               linklev = linkobj(ftw);

               /* use (dev,ino) to generate a unique name for this object */
               sprintf(name,"n%d_%d_%d",ftw->statb.st_dev,ftw->statb.st_ino,
                       linklev >= 0 ? ++N : 0);
        }

        /* generate the draw node statement */
        shape = ftw->local ? "Diamond" : ftw->info == FTW_F ? "Ellipse" :
               ftw->info == FTW_NS ? "Plaintext" : "Box";
        printf("draw %s as %s label \"%s\";\n",name,shape,ftw->name);

        if(ftw->level > 0)
        {      /* generate the (parent,child) edge statement */
               sprintf(parent,"n%d_%d_0",
                       ftw->parent->statb.st_dev, ftw->parent->statb.st_ino);
               printf("edge from %s to %s;\n",parent,name);
        }

        if(linklev >= 0)
        {      /* draw the link edge and prune the search */
               sprintf(link,"n%d_%d_0",
                       ftw->statb.st_dev,ftw->statb.st_ino);
               if(linklev == ftw->level)
```

```
                printf("same rank %s %s;\n",name,link);
        printf("%sedge from %s to %s dotted;\n",
                ftw->level >= linklev ? "back" : "",name,link);
        ftw->status = FTW_SKIP;
    }
    return 0;
}
```

For comparison, below is a **tw** script that is computationally equivalent to *draw*

```
    tw -P -d $1 -e "
    begin:
        printf('.GS\n');
    end:
        printf('.GE\n');
    action:
        if (type == LNK)
        {
                status = FOLLOW;
                local = 1;
        }
        else
        {
                printf('draw n%d_%ld_%d as ', dev, ino, visit);
                if(local)
                        printf('Diamond');
                else if(type == REG)
                        printf('Ellipse');
                else if(type == NS)
                        printf('Plaintext');
                else    printf('Box');
                printf(' label \"%s\";\n', name);
                if(level > 0)
                        printf('edge from n%d_%ld_0 to n%d_%ld_%d;\n',
                        parent.dev,parent.ino,dev,ino,visit);
                if(visit == 0)
                        visit = level+1;
                else
                {
                        if(visit == level + 1)
                        printf('same rank n%d_%ld_%d n%d_%ld_0;\n',
                        dev,ino,dev,ino,visit);
                        printf('%sedge from n%d_%ld_%d to n%d_%ld_0 dotted;\n',
                        (visit >= level + 1) ? '' : 'back',
                        dev,ino,dev,ino,visit);
                        status = SKIP;
                }
        }
    "
```

Figure-6 shows the generated DAG code corresponding to the hierarchy shown in Figure-3. In DAG, nodes are identified by unique strings and edges are defined by either an `edge` or a `backedge` statement. The `draw` statement defines the shape used to draw a node or collection of nodes and the labels for such nodes if they are different from their names.

```
.GS
draw n2324_141316_0 as Box label "/home/kpv/Src/TEST";
draw n2324_24634_0 as Box label "dir1";
edge from n2324_141316_0 to n2324_24634_0;
draw n2324_24637_0 as Ellipse label "file1";
edge from n2324_24634_0 to n2324_24637_0;
draw n2324_122913_0 as Box label "dir2";
edge from n2324_141316_0 to n2324_122913_0;
draw n2324_24637_1 as Ellipse label "file2";
edge from n2324_122913_0 to n2324_24637_1;
same rank n2324_24637_1 n2324_24637_0;
backedge from n2324_24637_1 to n2324_24637_0 dotted;
draw n2324_141316_2 as Diamond label "dir3";
edge from n2324_122913_0 to n2324_141316_2;
backedge from n2324_141316_2 to n2324_141316_0 dotted;
.GE
```

Figure 6. DAG description of a file hierarchy

Support of the ISO-9660/HSG CD-ROM File System in HP-UX

Ping-Hui Kao
Bill Gates
Bruce Thompson
Dale McCluskey

HP-UX Development Lab
Hewlett-Packard Corp.
3404 E. Harmony
Fort Collins, CO 80525

Abstract

The *ISO-9660* standard[1] and High Sierra Group (*HSG*) working paper[2] describe file system formats for publication and distribution of information on *CD-ROM* (Compact Disk Read Only Memory) media. This paper describes Hewlett-Packard's design and implementation of support for the ISO-9660/HSG CD-ROM file system (*CDFS*) in the HP-UX kernel. After mounting a CD-ROM which adheres to the standard, files on the CD-ROM are accessible through the normal system calls and commands allowing users and application programs to take advantage of the high capacity and low duplication cost of this medium without the need for any special programmatic interface.

The first section gives a brief overview of CD-ROM history and the ISO-9660/HSG file system format. The design goals, design philosophy, and testing strategy will then be presented followed by some examples of how this new feature will be used in HP-UX systems. The paper will conclude with a discussion of the results of the project and possible enhancements to the design.

1. Introduction

The CD-ROM is easily the most cost effective and versatile electronic distribution standard ever developed. It has many merits: large capacity (550 Mb), longevity, low cost, multi-media (audio/video) capability, read only, and random accessibility. CD-ROMs are ideal media for distributing data such as software, databases (e.g., government data, CAD databases), etc. Because of this, CD-ROMs can and will play a major role in the areas of software distribution, information retrieval, education, etc. Therefore, after the format was standardized, the decision was made to add CD-ROM support to the HP-UX operating system.

CD-ROMs and the ISO-9660/HSG standard have become well established in the MS-DOS[1] world. The contribution of this development effort is to bring this standard medium and operating system independent file system transparently into the UNIX[2] world.

Currently, there are large amounts of PC software distributed on CD-ROMs. Microsoft[3] MS-DOS CD-ROM extensions[5,6] provide the capability to access files on CD-ROMs. CDFS is an implementation of the MS-DOS CD-ROM extensions on HP-UX systems. More than 95% of the PC software should be able to be directly run on DOS-coprocessors or DOS emulation on an HP-UX system that supports the CD-ROM file system. With CDFS, applications running on PCs are brought one step closer to UNIX environment.

2. CD-ROM history/background

The CD-ROM technology was directly leveraged from the CD audio standard. The media specifications for audio is shared by CD-ROM. The bits are recorded on the media in one long spiral which can store up to 550 megabytes of data. Leveraging the technology from the audio standard does create two limitations. First, for maximum utilization of storage capacity and to maintain a constant data rate, the rotational speed of the disk is changed as the head is moved radially. This causes the seek times to be relatively long compared to other random access storage devices because of the need to change the spindle motor speed. Second, the I/O transfer rate (150kb/sec) is limited by the sampling rate required for digital audio.

One advantage of leveraging the CD audio standard directly is that CD-ROM disks can be duplicated using the same equipment used for audio CDs. The cost of producing disks usually amounts to a one time mastering charge and a cost per disk produced. These have been as low as $1500 for mastering and $2 per disk in the past year. Although the price per disk might not go down further, the price for mastering is expected to drop in the coming years.

3. CD-ROM file system (ISO-9660/HSG)

3.1 Data Layout

The overall data layout on a CD-ROM can be represented as in figure 1. There are typically four sections delimited by double horizontal lines in the figure; only the *system area* and *volume descriptors* must occur in the order shown.

The *system area* section, whose content is not specified, consists of the first 16 2048-byte blocks on the media. This area is reserved for storing information to boot a system, secondary loader, encryption keys and other system data supplied by the disk preparer.

1. MS-DOS is a registered trade mark of Microsoft Corp.
2. UNIX is a registered trade mark of AT&T in the U.S. and other countries.
3. Microsoft is a registered trade mark of Microsoft Corp.

The *volume descriptor* section typically contains one primary volume descriptor and zero or more supplementary volume descriptors. Each descriptor describes the attributes and structure of a directory hierarchy on the CD-ROM. The list of volume descriptors is terminated by a *volume descriptor terminator*. With these volume descriptors, there can be multiple directory hierarchies on a single volume (i.e., a physical CD-ROM), or a single directory hierarchy may span multiple volumes.

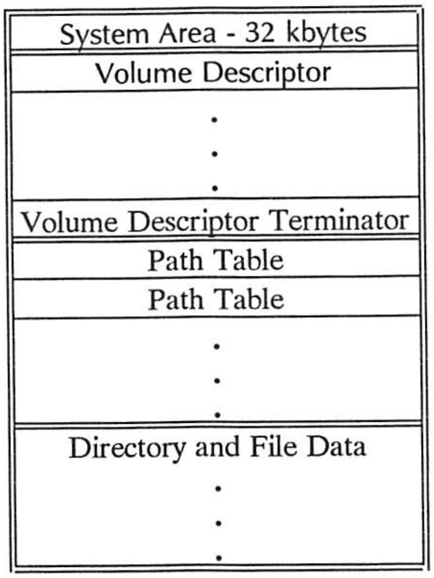


Figure 1: ISO-9660/HSG data layout

A *volume set* is a set of one or more volumes which are to be thought of as a unit. Each successive volume in the volume set updates or augments the data on the volumes preceding it.

The *path table* section contains all the path tables for all directory hierarchies on the CD-ROM. Each record in the path table contains information for the system to locate the directory it describes. The path tables do not have to be placed together as shown in the table above. They may be spaced out across the CD-ROM in whatever manner is acceptable to the data preparer for the CD-ROM. This is often done to minimize disk access times.

The *directory and file data* section contains all the directory and file data for all directory hierarchies on the CD-ROM.

The *directory* contains some number of directory records, each describes a directory or a file. The first directory record in a directory describes the parent directory of the directory; the second record describes the directory itself. Each record is layed out as follows:

Length of Directory Record
Length of XAR record
Location of file/directory
Size of file/directory
Recording Date and Time
File flags
File unit size
Interleave gap size
Volume sequence number
Length of file name
File name
System use area

Figure 2: ISO-9660/HSG directory record

A file contains data that can be recorded in interleaved mode for the performance reason. In this case, the file is divided into pieces called *file units* and recorded with *file unit gaps* between them. Also, a file can be broken up into *file sections*. Each file section is treated like a separate file in that each section gets its own directory record. All file sections for the same file must all share the same filename. Thus a file span over multiple volumes is possible.

An *extended attribute record* (abbreviated XAR) is an optional data structure specifying additional information about the file or directory with which the XAR is associated. An XAR contains information such as the owner and group ids, access permissions, and creation, modification, expiration, and effective dates and times. The XAR of a file or directory is recorded before the actual data of the file or directory. The most complicated form of a file section is as shown in figure 3.

XAR
File Unit Gap 1
File Unit 1
File Unit Gap 2
File Unit 2
. . .
File Unit Gap n
File Unit n

Figure 3: File section layout

4. Design Goals

The primary objective of the CD-ROM project was to provide easy access to this new medium so that applications including HP product distributions could take advantage of CD-ROM technology. The design goals for the project included the following:

- **Mounting:** The user must be able to mount ISO-9660/HSG file systems under HP-UX. The files on the CD-ROM are then accessible through normal HP-UX commands and system calls like those on the HP-UX native file system, *HFS*, a BSD derivative [3]. Additionally, the user must also be able to mount other file systems (HFS, *NFS* (Network File System), CDFS, etc.) under directories on the CDFS.
- **Networking:** Files on ISO-9660/HSG CD-ROM must be accessible via supported networking services including NFS and *RFA* (Remote File Access - an HP-proprietary network service for file accessing).
- **Application Programs:** An application program that is functional on a read only HFS file system must be functional on a CDFS file system unless it is dependent on special features or the format of HFS.
- **ISO-9660 vs HSG:** Although the ISO standard is the official standard, most CD-ROMs being produced today adhere to the HSG working paper. Any differences between these two formats must be hidden from the user by the kernel.
- **Adherence to HP's diskless schematics:** Unlike most "distributed file systems", one of the major features in HP's implementation of diskless workstations is that all diskless clients in a cluster have the same view of the files on all mounted file systems. This feature must be preserved with the addition of CDFS.
- **Performance:** The I/O transfer rate should be as close to a CD-ROM drive's maximum rate (150kb/sec) as possible when the benefits from buffer caching are excluded. The number of disk seeks must be minimized. File system buffering and path name caching must be used to help overcome the transfer rate and seek time (up to 1 sec) limitations of CD-ROM drives.
- **Configurability:** To minimize the size of the kernel, the user must be able to configure out the CDFS-specific code if the feature is not required.
- **File execution:** Programs recorded on ISO-9660/HSG formatted CD-ROM must be able to be executed directly. Executable files in demand-loaded form should be executable as well as others such as, shared text executable and regular executable.
- **Level of implementation:** The resulting implementation must conform to level 1 implementation, level 2 interchange according to ISO-9660. Basically, access is limited only to the file system described by the primary volume descriptor and to single-section files.
- **Quality of implementation:** The CDFS code must be of good quality and be easy to maintain. Techniques, such as *structured design*, should be used to help achieve this goal.

5. CDFS Design/Implementation

The structure of the HP-UX file system is based on the abstraction of file systems and file system operations referred to as the vnode layer, introduced by Sun Microsystems

Inc. This structure readily supports the addition of new file systems. This section will start with a brief discussion of the vnode layer. (For more details about the vnode layer, please refer to [4]). This will be followed by a detailed description of the implementation of CDFS and a discussion of the design considerations.

5.1 Vnode Layer

The vnode layer was previously added to HP-UX to allow support for non-HFS file systems (see figure 4). Given that it was already in place, it seemed logical to implement the new file system underneath the vnode layer.

The advantages of doing this are:

1. **Modularity**: The part of the kernel that supports CD-ROM file systems can be cleanly separated from the other parts of the kernel.
2. **Interoperability**: The CD-ROM file system can cleanly coexist with other supported file systems (e.g. NFS). CD-ROM files on an NFS server are made available to NFS clients.
3. **Integration:** Other file systems can be mounted on CD-ROM directories. This provides the capability of updating a directory in a CD-ROM by mounting a floppy disc on the directory. Also, additional CD-ROMs can be mounted under a directory to create a very large directory hierarchy; one that would exceed the capacity of a single disk. By mounting these CD-ROMs on different directories, user can have different configurations of the directory hierarchy.
4. **Resource sharing**: The CD-ROM file system can cleanly share system resources (e.g., buffer management, name cache, drivers, etc.) with other file systems.

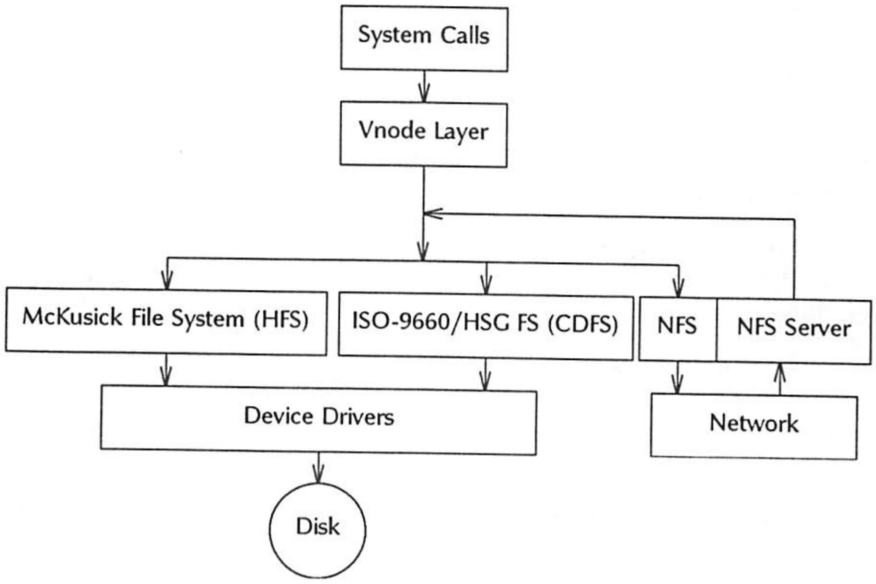


Figure 4: Vnode Architecture

At the vnode layer each file has an associated data structure called a vnode. This is a generic data structure used by all the supported file system types to describe attributes about the file. One entry in the vnode is a pointer which points to a file-system-dependent data structure. Any information needed by the specific file system implementation is stored here.

The higher-level layers of the kernel operate on vnodes; low level file-system-specific operations are encapsulated in separate modules. For example, the main part of the kernel is ignorant of CDFS; that knowledge is encapsulated in the CDFS code.

5.2 HFS/inode

In an HFS file system, each file has an associated data structure, called an inode, which resides on the disk. Each inode in a file system has a unique number associated with it. When the inode is needed, it is read in and supplemented with other information so that the system can use it to perform operations on the file.

5.3 CDFS/cdnode

In the case of CDFS, the file system dependent pointer in the vnode points to a *cdnode* that is created to store information similar to an inode but specific to CDFS. Unfortunately, inode-like structures with unique numbers do not exist in the ISO-9660/HSG file system. Cdnode information must be created using directory entries and optional *XAR*s (eXtended Attribute Records) as defined by the ISO standard. If a file does not have an XAR associated with it, some of the fields in a cdnode will contain default values. (This is explained in more detail later.)

A unique number required to identify a cdnode, similar to an HFS *i-number*, needs to be created since it doesn't exist on a CD-ROM. Our solution was to use the disk address of the directory entry; this uniquely identifies a file because the ISO-9660/HSG format does not support multiple links. (See section 5.6.5 for details on the directory cdnode number) In addition, this solution also provides us with the ability to locate the directory entry without any extra calculation.

5.4 Pathname Lookup

One operation of the kernel performs frequently is the resolution of pathnames. Name resolution is traditionally done one pathname component at a time, to provide for clean handling of mount points. With the advent of different file system types, traversing the pathname has become even more complicated.

Each vnode is defined by a unique file system type specifier and a set of services required by UNIX semantics. To perform a pathname lookup on a vnode, the pathname lookup function for that file system is called (e.g., nfs_lookup), returning the vnode for the next component. If a mount exists on this vnode, then changing file systems is indicated, perhaps to one of a different type. The vnode of the root directory for the new file system is obtained. This vnode may now contain different file system dependent functions than the previous vnode, which are then used as a new starting point to continue the process until all components have been parsed. Thus each file system is allowed to handle its own directory searches, and mounting of different file systems on arbitrary directories is possible.

Due to the long execution path and the slowness of disk seek, the scheme above could be very time-consuming. To help with this, the "directory name lookup cache" (DNLC, a feature that comes with the vnode layer) is used. It caches frequently used pathname elements and their vnode pointers, which significantly reduces the amount of redundant disk accesses made during pathname lookup.

The ISO-9660/HSG format provides a supporting data structure called a *path table* which is included for a similar purpose. The path table describes the entire directory structure of a file system. This is to allow traversing the entire pathname with one seek to avoid the lengthy seek time of CD-ROM drives. This method does not allow checking of mount points during the traversal. Also the path table can be very large, and could consume a large amount of the available memory if kept in main memory entirely. If the path table had to be referenced from the disk, much of its potential benefit would be lost. Although not ideal, most of the path table's performance gain is already provided by the *directory name lookup cache*. For these reasons, path tables are not used.

5.5 Backward Compatibility

One major design goal was to minimize the impact on application programs. The strategy to achieve transparent access to CDFS was to map, as much as possible, the characteristics of ISO-9660/HSG onto the standard UNIX semantics and characteristics. The first area of concern was the directory library routines since CDFS directory entries are quite different from those of HFS. Those routines call the *getdirentries* system call to obtain directory entries in a file system independent way. When *getdirentries* is used to read a CDFS directory, the fields of the directory entries are mapped into the format of a standard directory entry. Another potential problem area was the *stat* system call. To preserve the object code compatibility of existing compiled programs, the *stat structure* was not changed. Information about a file that maps well into the stat structure is passed back in it; other items specific to ISO-9660/HSG formats, such as *file unit size, interleave gap size*, etc., are dropped. A new system call was created to obtain the original data structure which contains those pieces of information that do not map well into the stat structure. Standard UNIX file attributes, such as *user id, group id*, and *permissions* are optionally specified in ISO-9660/HSG in an XAR. For files without an XAR, the *user id* and *group id* are simply set to an out of range number and permissions are set to 0555 (readable and executable by everyone as specified in the standard). The new system call *fsctl* is provided to retrieve information specific to any file systems. In this case, *fsctl* returns directory entries, XARs, parts of volume descriptors, etc. The reason why we rejected the idea of using *ioctl* in favor of *fsctl* was that *ioctl* is intended for control of special devices.

5.6 Other Design Considerations

During the design, the following obstacles had to be overcome:

5.6.1 Size of basic disk block

The buffer management of the kernel requires that all file system blocks be accessible in *DEV_BSIZE* (currently 1024 bytes) units. Unfortunately, the minimum size of an accessible CD-ROM disk block is 2048 bytes (a multiple of *DEV_BSIZE*) according to the CD-ROM standard. For read requests to a CD-ROM, care is taken in CDFS code to ensure that reads start on 2048-byte boundaries with sizes in multiples of 2048-byte blocks.

5.6.2 Smaller logical blocks

The logical block size of a CD-ROM can be 512, 1024 or 2048 bytes as determined by the data preparer. If the size of a logical block is less than 2048 bytes, the disk block containing the logical block must be read from the disk. Then only the logical block is copied into the user's buffer.

5.6.3 *Interleaving*

In an attempt to optimize access time and match an application's expected access patterns, files on a CDFS can be recorded in interleaved mode on a per file basis, and the size of a file unit and a file unit gap can be random as long as the sizes are multiples of sectors (2048 bytes). A well tuned routine, *cdfs_rd*, uses information such as, location of file, size of XAR, file unit size and file gap size, in the cdnode of the file to carefully calculate the location of each section of a file and concatenate pieces of data from different sections into blocks if necessary before passed them back to the callers. This routine also try to maintain the size of buffer to 8k bytes whenever appropriate so that buffer management can maintain its efficiency.

5.6.4 *Demand-paged exec*

One major challenge was to support direct execution of demand-paged programs on a CD-ROM. In the current implementation of HP-UX, virtual memory depends on files being physically divided into fixed size blocks (minimum of 4096 bytes for HP9000 300 Series), except the last block, and the disk address for each block is kept in the page table when the files are first *exec*'d. At the time of pagein, that disk address is used to read in the block by calling the relevant device strategy routine directly. Since the files on a CD-ROM can be recorded in interleaved mode and the size of file units can be any number of logical blocks, we cannot rely on the pagein routine to read in pages from the disk directly by calling the device strategy routine. Instead of the physical disk address, the offset into the file is stored in the page table. At pagein time, the *cdfs_strategy* routine uses this file offset to read in the page by calling *cdfs_rd* routine regardless of how the file is recorded. *Cdfs_rd* routine shields the fact that the portion of the file containing the page may not be contiguous on the disk.

5.6.5 *Directory cdnode number*

As mentioned above, the cdnode number is determined by the disk address of the directory entry. There is one difficulty with this approach. A directory can be described by a directory entry in its parent directory or the first directory entry of itself, aka ".", or the second directory entry, aka "..", of all its sub-directories. (See figure 5) Only one of the disk address of the directory entries described above can be chosen as the cdnode number for the directory. The use of directory entry for ".." entry was quickly ruled out by the fact that there can be many sub-directories which means they can not uniquely identify the directory. The choice between using "." and the directory entry in the parent directory was not obvious until the case of resolving the pathname ".." was considered. If the directory entry were used, at least three reads of different directories are needed to perform a pathname lookup of "..". The three reads are: 1) read of the ".." directory entry of current directory to locate the parent directory. 2) read of ".." of the parent directory to locate the grand-parent directory. 3) read through the grand-parent directory to locate the directory entry that matches the parent directory in order to use its address as the cdnode number. On the other hand, to obtain the cdnode number of ".." is much simpler with the choice of disk address of the directory entry of "." as the cdnode number. The only operation needed is to read the directory entry of ".." and from it the location of the parent directory can be calculated easily. Further, the cdnode number of the parent directory was stored in the cdnode when ever possible. The chance of having to read is greatly reduced.

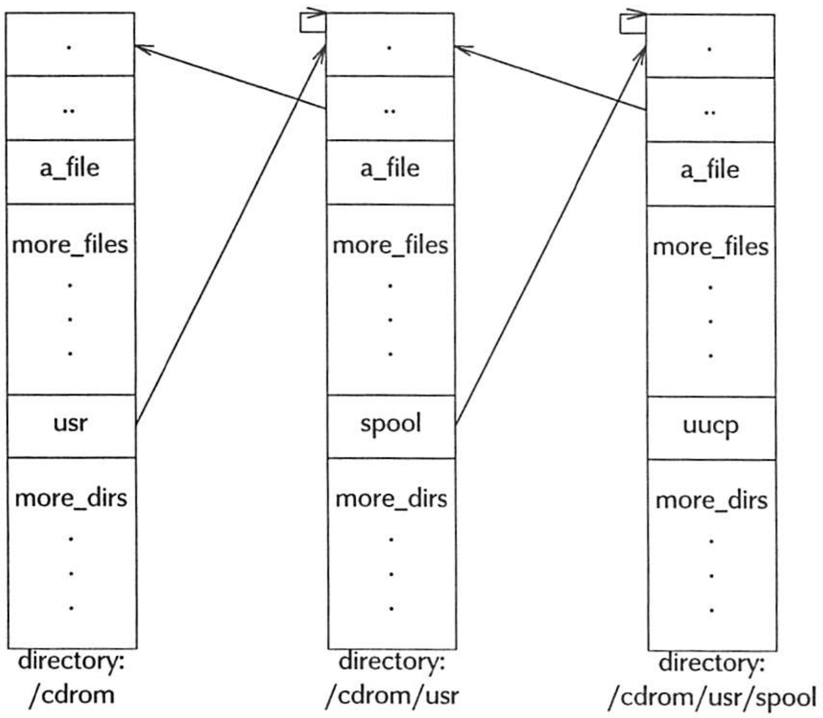


Figure 5: Directory Structure

5.6.6 Diskless protocol

HP-UX supports diskless clusters and a client's requests are passed via a light-weight protocol to the server. This protocol assumed there was only one kind of file system. A switch was added to this protocol to accept different kinds of file systems.

6. Applications

Several commercial CD-ROMs that contain MS-DOS software were purchased. This software includes:

1. Hewlett-Packard HP LaserROM
2. Microsoft Bookshelf[4]
3. Microsoft Programmers Library

4. Bookshelf is a trade mark of Microsoft Corp.

4. Microsoft Stat Pack
5. Microsoft Small Business Consultant
6. McGraw Hill: Science & Technology encyclopedia
7. Geovision[5]: Windows On the World
8. Ziff Davis: Computer Library
9. The PC-SIG Library

All of these were run on individual workstations equipped with DOS-coprocessor cards. With a simple mount, these applications were made available to users on local machines as well as remote machines via UNIX network services. For example, the PC-SIG Library CD-ROM contains applications which used to be distributed on 1000 floppy disks. Once this disk is mounted, all these applications become available to other systems with network access.

More CD-ROM applications that are native to UNIX systems will be more prevalent in the future to address the long standing need of installation, update, information retrieval, etc.

7. Testing and Validation

Several methods were used to test and validate the CDFS implementation.

1. Several commercially-available CD-ROMs were purchased to see if they could be accessed through CDFS. In selecting which CD-ROMs to purchase, the mastering companies used to produce the disks were determined and then at least one CD-ROM was purchased from each. In doing so, CDFS code was exposed to various interpretations of the standard.
2. The HP-UX kernel is subjected to nightly regression and verification testing via an automated test suite. Tests were added to this suite which mounted a CDFS volume and verified proper system call behavior. System calls were tested on both standalone and diskless configurations.
3. A test CD-ROM was produced which contained many items allowing testing of things not available on commercial CD-ROMs. This "test disk" contained several releases of HP-UX (to investigate the possibility of software distribution on CD-ROM), executables from the Series 300 and Series 800 (in particular, to test demand-loadable executables), huge files, small files, uncommon filenames, etc. The test disk was particularly useful in testing the demand-loadable executables, but because the CD-ROM manufactures could not create some of the more exotic data constructs (such as extended attribute records and interleaved files), testing of these constructs was not possible.
4. Because there are many possible data constructs which are not widely used in the industry today (such as interleaving, multi-section files, associated files, extended

5. Geovision is a trade mark of Geovision Inc.

attribute records, etc.), there was no way to test the paths in CDFS code which handled these cases. To solve this, a CD-ROM image generator, called cdgen, was written.

Cdgen takes a list of files and creates a CD-ROM image from them. This image is then written to either an HP-UX file or a hard disk. The hard disk can then be mounted and treated as a CDFS volume. Cdgen supports the following standard data formats and constructs:

- HSG or ISO-9660 format;
- multiple directory hierarchies per volume (primary and supplementary volume descriptors);
- multiple volume descriptor set terminators;
- interleaving;
- extended attribute records;
- multi-section files;
- associated files;
- system use, system area, application use, and escape sequence data in all constructs that provide for them.

Cdgen was first used to test rather obscure corner cases (such as interleaved demand-loadable executables, directories ending exactly on a sector boundary, zero-length files, system use data in "." and ".." directory records, and so on). Later, automated tests were written incorporating cdgen, which were then added to the nightly kernel test suite.

5. Code-reading techniques were used to verify the CDFS code. Both author and non-author code-reading was used.

Using the techniques above, a thorough job was done in verifying the robustness and solidity of CDFS under HP-UX.

8. Future Work

For the first release, level two data interchange and level one implementation according to ISO-9660 standard are supported. There is one restriction: there is a limit of one physical CD per volume set.

Support for the following is being considered:

- Level 2 implementation: allow mounting the file system described by a supplementary volume descriptor:
- Level 3 data exchange: support multi-section files.
- Multi-disk volume set: support volume sets that consist of more than one CD-ROM.
- Associated files: make associated files available to users.
- Hidden files: Make hidden files invisible to users.

- Dated files: Make dated files available only after effective date and before expiration date.
- MS-DOS CD-ROM extension *INT 2F* functions: Although *fsctl* covers all functions in MS-DOS CD-ROM extension *INT 21* functions and some of *INT 2F* functions (e.g., get copyright file name), support for other *INT 2F* functions (e.g. absolute disk read) could be added.
- Removable media: Like floppy disks, CD-ROM disks can be easily removed from the drives while they are mounted. Although this is not a problem specific to CD-ROMs, systems should be made to avoid potential system crash by user's mistakes.
- Change of default ownership and permission: Currently for files without an XAR, the *owner id* and *group id* of files are set to an out of range number and the *permissions* are set to 0555 as per ISO standard. A few alternatives are being considered:

  1. Use the *user id*, *group id* and *permissions* of the directory that is mounted on.
  2. Provide options to the *vfsmount* system call so that these values can be set when mounting the disk.
  3. Add a feature to *fsctl* to allow setting of these values.

  Clearly, these alternatives give users more flexibility. Because of the fact the ISO standard implies the default for permission should be *readable and executable by any user*, none of these alternatives were implemented.

9. Conclusion

All design goals were achieved and the delivered performance matches the speed of the underlying device. The structured design techniques made some sections of CDFS code very modular. Pathname lookup especially is much simpler and more modular than its counterpart in HFS, even though the CDFS directory structures are more complicated. Although the I/O rate is very dependent on the physical layout of the data on a CD-ROM, it was measured at 149.9kb/sec (the maximum for a the CD-ROM drive is 150kb/sec) without the help of the buffer cache. All non-file-system dependent commands and the CD-ROM applications, that do not directly control the device drivers, from independent software vendors tested up to now run without changes. The quality of this system is greatly improved after the last few corner case errors were identified by the images created by cdgen. Overall, the team is very pleased with the quality of the system and the fact customers can use this new feature as a foundation for their applications.

References

1. National information Standards Organization, "The Working Paper for Information Processing, Volume and File Structure of Compact Read Only Optical Discs for information Interchange", May, 1986
2. International Organization for Standardization, "Information Processing - volume and file Structure of CD-ROM for Information Interchange", ISO-9660: 1988(E) Edition",

1988

3. M.K. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX", ACM TOCS, 2, 3, August 1984, pp 181-197

4. S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", USENIX Conference Proceedings, Atlanta, Georgia, (Summer 1986)

5. Microsoft Inc., "Microsoft MS-DOS CD ROM Extensions Function Requests", Jan. 1988

6. Microsoft Inc., "Microsoft MS-DOS CD ROM Extensions Hardware-Dependent Device Driver Specification", Feb. 1988

Simple and Flexible Datagram Access Controls for Unix-based Gateways

Jeffrey C. Mogul
Digital Equipment Corporation Western Research Laboratory

Abstract

Internetworks that connect multiple organizations create potential security problems that cannot be solved simply by internal administrative procedures. Organizations would like to restrict inter-organization access to specific restricted hosts and applications, in order to limit the potential for damage and to reduce the number of systems that must be secured against attack. One way to restrict access is to prevent certain packets from entering or leaving an organization through its gateways. This paper describes simple, flexible, and moderately efficient mechanisms for screening the packets that flow through a Unix-based gateway.

1. Introduction

Internetworking has greatly improved communication between administratively distinct organizations, linking businesses, schools, and government agencies to their common benefit. Unfortunately, internetworks that connect multiple organizations create potential security problems that cannot be solved by the mechanisms used within organizations, such as restricting physical access. In particular, interconnection at the datagram level is an ''all or none'' mechanism, allowing outsiders access to all the hosts and applications of an organization on the internetwork. To avoid penetration, every host within an organization must be made secure, no small feat when it involves tens of thousands of poorly-managed workstations.

We would like to be able to restrict inter-organization access to specific hosts and applications. Doing so limits the potential for damage, and reduces the number of systems that must be secured against attack. One approach is to place an application-level gateway between the organization and the internetwork, eliminating packet-level access but supporting a small set of approved and presumably bullet-proof applications. Typically, these include electronic mail, name service, and perhaps remote terminal access.

This approach is painful because it requires writing application gateway software for each approved application, and because it may be too draconian for some organizations. It also significantly reduces performance.

A more flexible way to control access is to prevent certain packets from entering or leaving an organization through its gateways. This allows greater flexibility than an application-level gateway, although as with any power tool it also requires greater vigilance. Various commercial gateway systems provide such a mechanism [2, 27] and it has also been treated in the literature [10].

Although it is not a good idea to use a Unix® system as an internetwork gateway, the popularity of 4.2BSD Unix (and its successors and derivatives), coupled with bureaucratic inertia, has led numerous organizations to use Unix-based gateways. The mechanisms described in this paper allow users of Unix-based gateways to impose packet-level access controls without major changes to existing software. Perhaps more important, by separating mechanism from policy [17] so that arbitrarily precise access control policies may be developed as ordinary Unix user processes, these mechanisms support fine-tuning and experimentation that would be difficult using commercial gateway products.

Section 2 of this paper sets out the background and goals of this project, and describes previous work in the field. Section 3 presents the design and implementation of a simple, flexible, and efficient modification to the Unix kernel. Sections 4 and 5 then describe two different user-level daemons that make use of this modified kernel to provide packet-level access control. Section 6 discusses how the new mechanism may be used for purposes besides access control.

2. Background and Goals

The Internet Protocol (IP) suite [15] is today the dominant means of connecting disparate organizations into an internetwork. Virtually all of the practical and experimental work on datagram access controls has been done using IP protocols. Although the examples in this paper do assume the use of IP, much of the mechanism (especially the kernel support) should be applicable to any similar protocol suite, including OSI [30].

All datagrams in an IP internetwork carry an IP header [24], which includes the source and destination host addresses. In general, this is all that an IP gateway may assume about a datagram, so one might choose to restrict access on a host-by-host basis. (Since IP network numbers can be extracted from IP host addresses, and since network numbers can often be identified with specific organizations, an IP-level mechanism might also restrict datagrams based on source or destination network number.)

In fact, however, almost all information is carried by transport protocols layered above IP. Primarily, these include TCP [25] for reliable byte-stream applications (mail, file transfer, remote terminal access), and UDP [23] for request-response protocols (name service, routing, the NFS file access protocol [28]). Certain control information is carried by the ICMP protocol [26]. One may wish to require or restrict the use of such higher-level protocols, which can be done based on information in the IP header.

Further, the TCP and UDP protocols incorporate the concept of a ''port,'' identifying an endpoint of a communication path, and these protocols support the concept of ''well-known'' ports. For example, a TCP remote terminal access connection will always be addressed to well-known port 23 at the server host. In some cases, it may be useful to require or restrict access to specific ports; the access-control mechanism in this case would have to examine the higher-level header, since the source and destination ports do not appear in the IP header.

2.1. Policy and Mechanism

There is a wide range of access control policies from which to choose. One goal of an access control mechanism should be to allow each organization to choose its own policy, and to change its policy (perhaps quite frequently). In other words, it pays to separate the *mechanism* for forwarding packets from the *policy* that decides what should be forwarded. Although the underlying concept has been known for a long time, the term *policy/mechanism separation* was invented by the designers of the Hydra system [17], who established it as a ''basic design principle ... of a kernel composed (almost) entirely of mechanisms.'' Policies were embodied in user-level processes, thus improving flexibility while keeping the kernel simpler and presumably more reliable.

The Unix kernel, on the other hand, contains most of the policy functions of the operating system. There are a few exceptions; for example, in 4.2BSD Unix, the disk quota mechanism in the kernel receives quota information from user-level processes, and the network routing table is maintained by a user-level process. Still, in the Unix model the kernel makes the decisions.

The system described in this paper adheres to the Hydra model: a simple, general mechanism inside the kernel ''asks'' a user-level process to pass judgement on every packet that is to be forwarded. The kernel makes no access control decisions; rather, it provides the packet header to the user process, which then tells the kernel whether or not to forward the packet. The kernel mechanism is simple (it took about one day to code and test) and robust (a failure of the policy module should not result in a system crash, or indeed in any consequential failure save a temporary suspension of packet forwarding). Further details on the kernel mechanism are given in section 3.

Once this kernel mechanism is in place, it is easy to experiment with policy modules implemented as normal Unix user processes. Section 4 describes an implementation based on a daemon process that checks each packet against the criteria specified in a configuration file. This program is able to filter based on arbitrary criteria, including transport-level header information.

An alternative design is sketched in section 5. In this design, a user-level process would implement a *visa* protocol [10]. In visa protocols, there is no configuration file at the gateway; rather, the source host is required to attach a cryptographically-secured mark to the datagram header, proving to the gateway that this datagram is authorized to be forwarded. Authorization is done by an ''access control server'' distinct from the gateway; thus, visa protocols employ an additional level of policy/mechanism separation.

One subtle (but explicit) aspect of the IP architecture is that gateways are stateless packet switches, not required to maintain any history of previous packets [4]. The policy modules described in this paper can accommodate a certain amount of gateway state (see section 4.3), but may not support a protocol requiring significant gateway state. This is because so little information is passed between the gateway function in the Unix kernel and the decision-making function in the user process; neither function has access to the history of the other. This is not a serious problem for the IP protocol.

2.2. Related Work

It has long been recognized that an organization may protect itself against unwanted network connections by blocking them in its gateways. One simple approach is to remove from a gateway's routing tables routes to specific networks, thus making it impossible for a ''local'' host to send packets to them. (Most protocols require at least some bidirectional packet flow even for unidirectional data flow, so breaking the route in only one direction is usually sufficient.) This approach, of course, does not work when the point is to permit access to some local hosts but not others.

Most (perhaps all) commercially-available gateway systems now provide the ability to screen packets based not only on destinations, but on sources or source-destination pairs. For example, the Proteon p4200 gateway [27] allows the manager to specify access based on pairs of IP addresses; each address is combined with a specified mask before comparison, so that the pairs may refer to networks or subnets instead of specific hosts.

Gateway products from cisco Systems [2] support a more complicated screening scheme, allowing finer control over source or destination addresses. For example, one could deny access to all but one host on a particular network. The cisco gateways also allow discrimination based on IP protocol type, and TCP or UDP port numbers.

Unlike access controls based on gateway-resident configuration tables, *Visa* protocols [10] control the path between specific pairs of hosts. Moreover, visa protocols provide some protection against forged datagram headers, by proving that the source address of a packet is genuine. Visas thus protect against malefactors within the local organization, as well as those outside, and because policy decisions are made by a server distinct from the gateway, one can employ an arbitrarily complex policy without fear of overwhelming the gateway. In spite of the advantages of visa protocols, they require explicit support from host implementations and increase the per-packet effort at gateways, so they are only in experimental use [9].

The kernel-resident mechanism described in section 3 owes an intellectual debt to the ''packet filter'' mechanism [21] used to give user processes efficient access to arbitrary datagrams. In the packet filter, the kernel applies user-specified criteria to received packets before demultiplexing the packets to the appropriate process. This is the reverse of the situation described in this paper, where decisions are made in user processes and consumed by the kernel, but the principle of doing the more complex job at user level remains the same.

3. Kernel support

The mechanism described in this section is called the *gateway screen*. It has been experimentally implemented in the context of the Ultrix™ 3.0 operating system. Since the IP forwarding code in Ultrix is nearly identical to that in 4.3BSD Unix, this mechanism should port to nearly any 4.2BSD-derived kernel with only minor modifications. Porting it to unrelated Unix kernels, or to other operating systems, may require changes in various details. The presentation in this section assumes the use of a 4.2BSD-derived kernel.

3.1. Overview

When an IP packet is received by the kernel, it is first processed by the ***ip_intr()*** procedure, which determines if the packet is meant for "this host." If not, then the packet is passed to the ***ip_forward()*** procedure, which determines whether and how to forward the packet to its ultimate destination. The ***ip_intr()*** procedure runs as an interrupt handler, not in the context of a specific process.

The gateway screen intercepts packets just before they are passed to ***ip_forward()***. These potentially-forwardable packets are placed on a queue, and any processes waiting for this queue are awakened. (A process waits for this event by executing a particular system call.) When any such process wakes up, it removes a pending packet from this queue. The kernel extracts the datagram header from the packet, wraps this with some control information, and passes the result out to the user-level process. The system call then completes, allowing the user process to run and decide if the packet in question should be forwarded. The user process, through a subsequent system call, informs the kernel of its decision, and the kernel either drops the packet, or passes it on to ***ip_forward()***.

The kernel effectively acts as the "client" of a "server process" implemented at user level. The user process, however, makes the calls into the kernel, not vice-versa. One may view this as a relationship structured entirely of "up-calls" [3], with no "down-calls" at all. This structure requires the solution to certain problems (starvation, matching of requests with responses) not present in a more conventional client-server interaction, but it makes use of the synchronization, protection, and control mechanisms already provided by the normal system call implementation.

We can now look at the implementation in greater detail.

3.2. Interrupt-level functions

Not much processing is required at interrupt level. The ***ip_intr()*** procedure is modified to pass packets to the ***gw_forwardscreen()*** procedure, instead of directly to ***ip_forward()***. The ***gw_forwardscreen()*** procedure allocates a small control block to contain information about the packet, records the address of the packet buffer ("mbuf chain") in this control block, puts the control block on a queue of "pending" packets, and issues a ***wakeup()*** call. The control block includes fields for the packet arrival time and a unique transaction identifier, which are set in this procedure.

Since the kernel has little control over how fast the user-level process responds to requests, it would be unwise to allocate control blocks directly out of kernel dynamic memory, for fear of using it up. Instead, the gateway screen maintains a limited pool (currently 32 items) of preallocated control blocks; this not only avoids using up memory, but makes allocation much faster.

Since this limits the number of pending packets, we must have some policy to apply to packets arriving when our private free list is empty. Two policies suggest themselves: accept the new packet and drop an old one, or drop all new packets until some old ones have been processed. The former policy is more expensive to implement, and the latter policy conforms to the assumptions of various congestion-avoidance and congestion-control mechanisms[1]. Since dropping the incoming packet is both easier and "better," that is what is done. (Even when a packet is dropped, a ***wakeup()*** is still done, in case there are server processes sleeping.)

3.3. Programming interface

Before we look at the implementation of the system call interface, it is helpful to examine the alternatives for communicating information across the user-kernel boundary. There are two ways to do this in Unix: either the information is moved as the parameter (by-value or by-reference) of a system call, or it is moved via the I/O mechanisms over a file descriptor (I/O is done with system calls, of course, but these calls leave little room for improvisation).

---

[1]Nagle implies as much with his statement that, when facing buffer exhaustion, a gateway should "drop the packet at the end of the longest queue, since it is the one that would be transmitted last." [22] Jacobson's work [12] implies that gateways should give preference to the earlier packets in a multiple-packet window, since they are more likely to be retransmitted immediately once congestion is detected.

Note that for each packet, we need to communicate information out of the kernel, let the user process run, and then communicate the decision back into the kernel. We do not necessarily have to make two system calls per packet; the trick is to pass one decision into the kernel on the same system call that passes the following packet's information out of the kernel. If we want to use this trick, then we cannot use the normal I/O mechanisms (e.g., ***read/write*** or ***send/recv***).

Perhaps the cleanest approach would be to invent a new system call with one ''in'' parameter and one ''out'' parameter. Adding a new system call to the Unix kernel, however, requires changing a number of files within the kernel, as well as the system call interface library, and this seemed too painful.

The approach actually taken was to define a few new *ioctl* requests. Since an ioctl parameter can be ''value-result'' (both ''in'' and ''out''), we can use the ''one system-call'' trick, and since adding a new ioctl request requires no new code aside from where the request is dispatched, it is easy to make this change. The disadvantages are that an ioctl parameter can carry at most 127 bytes of data, and that the entire parameter is copied in both directions. The size limit is not actually a problem (the largest possible IP header is 60 bytes, and the largest reasonable TCP header is 24 bytes). The extra data copying is a slight disadvantage (compared to the ''new system call'' approach) but is less costly than having to do twice as many system calls.

The ***screen_data*** structure passed between kernel and user space for the SIOCSCREEN ioctl request contains a prefix of the packet (including at least the packet headers), a timestamp indicating when the packet was received, and a transaction identifier to be used in matching requests with responses. This is necessary because there is no ''connection'' (such as a file descriptor) established between the kernel and the server process, so there is no other way for the kernel to associate the decision passed in on one system call with the information passed out on a previous one. Figure 3-1 shows the layout of the ***screen_data*** structure.

```
/*
 * Some fields of this struct are "OUT" fields (kernel write,
 * user read), and some are "IN" (user write, kernel read).
 */

struct screen_data_hdr {
    short sdh_count;        /* length of entire record */    /* OUT */
    short sdh_dlen;         /* bytes of packet header */     /* OUT */
    u_long sdh_xid;         /* transaction ID */             /* OUT */
    struct timeval
        sdh_arrival;        /* time this pkt arrived */      /* OUT */
    short sdh_family;       /* address family */             /* OUT */
    int sdh_action;         /* disposition for this pkt */   /* IN */
                            /*      see defs below      */
};

/* Possible dispositions of the packet */
#define SCREEN_ACCEPT   0x0001  /* Accept this packet */
#define SCREEN_DROP     0x0000  /* Don't accept this packet */
#define SCREEN_NOTIFY   0x0002  /* Notify the sender of failure */
#define SCREEN_NONOTIFY 0x0000  /* Don't notify the sender */

/* Screening information + the actual packet   */

#define SCREEN_MAXLEN 120       /* length of struct screen_data */
#define SCREEN_DLEN (SCREEN_MAXLEN - sizeof(struct screen_data_hdr))

struct screen_data {
    struct screen_data_hdr sd_hdr;                          /* IN/OUT */
    char sd_data[SCREEN_DLEN];  /* pkt headers */           /* OUT */
};
```

Figure 3-1: Layout of the *screen_data* structure

The lack of a connection creates some complexity in the kernel implementation, but it avoids the greater complexity of creating and managing connections, and in particular avoids the need to garbage-collect connections belonging to dead processes.

The structure passed between kernel and user space also contains one field that is set by the user process, to indicate whether or not the packet should be forwarded, and if the packet is rejected, whether or not to notify the source host. (The ICMP protocol provides a means for a gateway to notify a host that a packet has been dropped; unfortunately, there is no "Access Violation" message, so we must make do with an approximation such as "Host Unreachable."[2])

In addition to SIOCSCREEN, new ioctls are defined to turn screening on or off (SIOCSCREENON) and to get statistics information (SIOCSCREENSTATS). Only the SIOCSCREENSTATS request is available to unprivileged processes; processes must be running as the super-user to execute the other requests.

Figure 3-2 shows the complete source of an simple daemon program that "decides" to reject all packets.

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <net/ip_screen.h>

main() {
    int s; struct screen_data scdata;

    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
        { perror("socket"); exit(1); }

    scdata.sdh_xid = 0; /* start with garbage transaction id */
    while (1) {
        if (ioctl(s, SIOCSCREEN, (caddr_t)&scdata) < 0)
            { perror("ioctl (SIOCSCREEN)"); exit(1); }
        printf("dropping pkt, transaction %d\n", scdata.sdh_xid);
        scdata.sdh_action = SCREEN_DROP;
    }
}
```

Figure 3-2: Example program that rejects all packets

3.4. Process-context functions

When the user process issues the SIOCSCREEN request (over a file descriptor returned by the ***socket()*** system call), control in the kernel passes to the ***ifioctl()*** procedure. This is the only place besides ***ip_intr()*** that must be modified to support the gateway screen. This procedure simply transfers control to a procedure called ***screen_control()*** if any of the gateway screen ioctl requests are made.

When viewed as a complete system call, the SIOCSCREEN request starts by copying a decision into the kernel, sleeps waiting for a new packet, and then copies new information out of the kernel. Since there is no connection between what happens before and after the sleep, it is easier to describe a complete cycle starting with the sleep rather than starting with the system call.

Each packet in the pending queue is either "claimed" or "unclaimed." A claimed packet is marked with the process ID of a process that has been given this packet to act on. When the process awakens from its sleep, it checks the queue of pending packets to see if there are any that have not yet been claimed. The pending-packet queue is managed first-in, first-out to avoid unfairly delaying any packet. If no unclaimed packets exist, the process returns to sleep. Otherwise, the packet is marked with the current process ID. A prefix of the packet, and other information from the control block, is copied into a ***screen_data*** structure, and the ioctl request com-

[2]The "Blacker" system [6], which provides military-style security by interposing cryptographic hardware between secure hosts and an insecure backbone network, does define an appropriate ICMP code, but no commonly-used host software recognizes it.

pletes in the normal way; since the kernel "knows" that the ioctl parameter is value-result, the ***screen_data*** structure is copied out to the user process.

Once the user process has made its decision, it sets the appropriate bits in the ***screen_data*** structure and reissues the SIOCSCREEN request. Again, the kernel knows to copy the ***screen_data*** structure into the kernel, and control is passed to the ***screen_control()*** procedure. At this point, the kernel must match the decision in this SIOSCREEN request to one of the packets on the pending-packet queue.

Since the ***screen_data*** structure contains the unique transaction identifier stored in the packet control block, the kernel simply searches the pending queue for a packet with the right transaction identifier. If none are found, then the user process has made a mistake (or is making its first system call) and no further action is taken. If, during the search, packets are found that are claimed by the current process, but that do not have the right transaction identifier, then the user process has failed to follow first-in, first-out order; these packets are removed from the queue and simply dropped.

Assuming that a matching packet is found, if the decision encoded in the ***screen_data*** structure is positive, then the packet is passed to ***ip_forward()***, as if it had come directly from ***ip_intr()***[3]. The ***ip_forward()*** procedure may of course decide not to forward the packet (for example, because no route exists) but this decision is made independent of access control.

If the user process instead decided against forwarding the packet, it is simply dropped. If the user process requested that the sender be notified, the ***icmp_error()*** procedure is called with the appropriate arguments, causing an ICMP Host Unreachable message to be sent.

At this point, the cycle is complete. The packet control block is put back on the private free list, and the user process is put back to sleep (after checking the pending-packet queue to make sure that no additional packets are waiting).

Earlier it was pointed out that there are a limited number of packet control blocks. Since it is possible for a user process to claim a packet and then die without providing a decision, the pending-packet queue could fill up with junk. Therefore, if when the pending-packet queue is searched and no unclaimed packets are found, and if the private free list is empty, we make the assumption that something is wrong. All packets older than a threshold age (for example, 5 seconds) are simply removed from the queue and dropped. If their "owning" processes are actually still alive and subsequently do render a decision, this causes no further problems. Thus, if a large number of processes do fail, the worst outcome is that the gateway stops forwarding packets for several seconds (provided that additional screening processes exist or are restarted).

3.5. Protocol-independence

None of the code within the kernel implementation of the screening module makes any assumption about the content of the packets being handled (including the layout of the packet headers). The only protocol-dependent actions required are the calls to either forward a packet or to send an error notification. These procedures are called indirectly, through pointers stored in the control block that were provided by the protocol-specific code that called ***gw_forwardscreen()***. To add screening for a new protocol family, one need only supply protocol-specific forwarding and error functions, and insert a call to ***gw_forwardscreen()*** in the appropriate place.

Since it is necessary for the user process to know which kind of packet it is processing, the protocol-specific module also provides a type code (the "address family": AF_INET for IP packets) that is stored in the protocol control block and passed to the user process (see figure 3-1). A process can "request" to receive packets of only one family by setting the `sdh_family` field when it passes a ***screen_data*** structure to the kernel.

[3]There is a subtle difference: normally, when *ip_intr()* calls *ip_forward()* it does so at an elevated interrupt priority level (IPL), but the gateway screen calls *ip_forward()* at low IPL. This does not seem to be a problem, but one must be careful to ensure that *ip_forward()* does not assume it is called at any particular IPL.

3.6. Performance

In order to get an idea of the performance of the kernel portion of the implementation, we can look at the limiting case of a minimal user-level daemon: for example, the program in figure 3-2, changed to accept all packets without examining them.

The most interesting characterization of performance is the increment in delay over an unmodified Unix-based gateway. (It is much easier to measure round-trip delays rather than one-way delays; we must assume that the underlying one-way delay is about half of the total.) This increment is easily measured using the ICMP Echo protocol, which is especially convenient because all the processing in the "echo server" host is done in the Ultrix kernel, reducing the variance in delay.

The performance in this case depends mostly on the cost of transferring data and control between kernel and user contexts; that is, system call overhead dominates the cost of code within the gateway screen implementation. The entire pending-packet queue is searched once per system call, so to some extent the length of that queue affects performance. Thus, since the cost of that search is linear in the queue length, two measurements suffice to define the performance of the kernel implementation: the incremental delay when the pending-packet queue is empty, and the incremental delay when that queue is artificially kept nearly full.

Table 3-1 shows the measured round-trip and calculated one-way delays for several gateway implementations: an unmodified Ultrix kernel, the gateway screen with an empty queue, and the gateway screen with artificial garbage entries in the queue. The experimental setup consisted of a gateway, based on a MicroVax™ 3500 (about 2.7 times as fast as a Vax™-11/780), connecting two Ethernets [8], with an echo client host on one Ethernet and an echo server host on the other Ethernet. Packets contained 56 bytes of data, in addition to 42 bytes of Ethernet, IP, and ICMP headers. The measurements reflect average delays over a large number of trials.

Time in milliseconds			
Version	Round trip	One way	Added delay
No screen	8	4	
Empty queue	10	5	1
Full queue	14	7	3

Table 3-1: Performance with minimal user-level daemon

One measurement was also made with the client and server on the same Ethernet, with no intervening gateway; this gave a round-trip time of 3 milliseconds, or a one-way time of 1.5 milliseconds.

The "Added delay" column in table 3-1 shows the increment in one-way delay over an unmodified Ultrix-based gateway. For this hardware, the gateway screen delays each packet by about 1 millisecond, which is consistent with the cost of doing a system call.

The "Full queue" case in the table reflects a situation where the length of the pending-packet queue is artificially maintained at 500 packets. The additional delay imposed by this queue was about 2 milliseconds, or about 4 microseconds per entry. Normally, this queue is limited to 32 packets, but at that length the effect (estimated to be 160 microseconds per packet) is too small to be measured. Note that as long as the packet rate remains below overload (about 200 packets/second) the queue will remain nearly empty.

One other measure of an implementation is its size. Aside from a few lines of code added for linkage from other modules, the gateway screen implementation consists of 436 lines of heavily commented code (and 140 lines of header file). Compiled for the Vax, this results in 1512 bytes of object code, and less than 1 Kbyte of data storage is required at run time.

3.7. Further work

Although the ioctl-based programming interface makes it quite easy to integrate the gateway screen into the Unix kernel, it is not as efficient as one might like. The screening function could be embodied in a new system call that copied the packet header information only in one direction.

It is also possible to reduce the system call count even further, by batching several packets and decisions together for one system call. Batching has been shown to be profitable in a similar application [21]. When the load is low, the pending-packet queue will seldom hold more than one packet, and the batch size will be 1, but at high loads, several packets may arrive before the user-level daemon process can be scheduled (packet interrupts having higher priority than user processes). Thus, as the system approaches overload, batch size increases, and the system call overhead per packet decreases; this is precisely the behavior one wants.

```
gateway_screen(packet_data, packet_count, decision, decision_count)
struct screen_data *packet_data;          /* pointer to buffer */
int *packet_count;       /* result returned by reference */
struct screen_data_hdr *decision;         /* pointer to buffer */
int decision_count;
```

Figure 3-3: Proposed new system call

Figure 3-3 shows how the programming interface might appear for a new system call supporting batched operation. The ***packet_data*** and ***decision*** parameters are vectors of one or more ***screen_data*** and ***screen_data_hdr*** structures, respectively, with their lengths specified by the ***packet_count*** and ***decision_count*** arguments, respectively.

Since for most access control policies, the forwarding decision for a packet is independent of any previously received packets, the gateway screen is a natural application for parallel processing. On a multiprocessor, one could run several copies of the user-level daemon process, each on its own processor. The current kernel implementation, since it uses raised interrupt priority level for synchronization, would have to be modified for use in a symmetric multiprocessing kernel; explicit locks would be needed for the pending-packet queue and the private free list. Contention should be relatively low, especially if the items on the pending-packet queue are individually locked when being manipulated, since locked items can simply be ignored when searching that list.

If the kernel kept a cache of recent decisions, as is done in the user-level program described in section 4.2, it could potentially avoid most of the transfers to user-level code, and so significantly improve performance. The trick is to choose a cache-match function; an incoming packet will almost never match a previously received packet in its entirety, so only a few selected fields can be used in the matching function. Not only would the choice of these fields be protocol-specific (and would probably involve several layers of protocol header), but it might also be policy-specific; the user-level policy process would want to specify the matching function. It does not appear feasible to implement a general-purpose mechanism in the kernel, although it might pay to provide a caching function for a few heavily used protocols, such as TCP.

4. A table-driven policy daemon

This section describes the design and implementation of *screend*, a table-driven policy daemon to make datagram access control decisions, to be used with the kernel mechanism described in section 3. This program uses only the most vanilla features of Unix (besides the gateway screen mechanism, of course) and so should be portable to any Unix-like system.

4.1. User interface

To use *screend*, one starts by generating a configuration file. This is a text file that describes the kinds of packets that should be accepted or rejected. The daemon program is then started, parses the configuration file, and then enters an infinite loop making packet-forwarding decisions according to the criteria in the configuration-file. If one wishes to change the criteria, one simply edits the configuration file and restarts the daemon process. (In principle, the daemon could notice that the file has been changed, but this might add unnecessary code to the performance-critical inner loop).

The complete grammar for the configuration file is rather involved, and is given in appendix I. To understand this section, one must understand some of the concepts that may be expressed by this grammar.

The main purpose of the configuration file is to specify the action (accept or reject) taken when a particular kind of packet arrives. Packets are identified by their IP source and destination address, the next level protocols they use (for example, TCP or UDP), and the source and destination ports, or ICMP type codes, if these apply[4].

A packet can thus be precisely specified by listing its source and destination addresses, its protocol type, and if applicable, its source and destination ports or ICMP type code. One can also leave any of these fields unspecified, or partially specified. For example, one may specify that packets from a particular host to any host at all, using any protocol at all, should be be rejected; this is one way to isolate that host from the internetwork. One may also partially specify an address by specifying not a host address but a network number, or perhaps a subnetwork number. Finally, one can specify that certain fields should not match for the entire specification to match. Most fields can be specified either numerically or symbolically.

Specifications are evaluated in the order that they appear in the configuration file. Thus, specific exceptions to a more general rule should appear earlier in the file. Also, note that to specify both directions on a path, one needs two rules. The actions taken, in addition to accepting or rejecting a packet, can include notifying the sender upon rejection, and logging the packet (useful for detecting breakins). Figure 4-1 shows some examples (these examples would not make sense for any single gateway, since the host names are chosen at random).

```
from host xx.lcs.mit.edu tcp port 3
        to host score.stanford.edu tcp port telnet reject;

between host sri-nic.arpa and any accept;

from net milnet to subnet 36.48.0.0 proto vmtp reject;

from net-not cmu-net tcp port reserved
        to net cmu-net tcp port rlogin reject notify;

from any icmp type echo to any accept log;
```

Figure 4-1: Example configuration file

There are two other kinds of rules that can be in the configuration file. First, one can specify a default action (normally rejection). Second, one can specify the network masks for arbitrary networks. This allows subnet specifications for non-local networks. Normally, a gateway is not supposed to know about the subnet structure of a distant network [20], but this information might be useful in deciding whether to forward a packet that originates someplace in an organization with multiple network numbers.

4.2. Implementation

Most of the complexity in the implementation *screend* is in the code that parses the configuration file and builds the internal data structures. The parser itself is a straightforward application of *lex* [16] and *yacc* [13]. All translations of symbolic values (such as host names) to numeric values are done during parsing; this is necessary for performance, although it means that if a host changes its address, the internal databases may reflect stale specifications.

The main loop of the daemon accepts a packet header from the kernel, extracts certain fields from the header, and matches the extracted fields against the internal representation of the configuration file. Since matching is the most time-consuming part of the inner loop, the choice of internal representation clearly affects performance.

[4]ICMP type codes are interesting because some ICMP messages can be harmful (for example, routing Redirect packets) while others are harmless (for example, Echo packets). Well, mostly harmless: a villain could use ''harmless'' packets to flood a victim's host.

Several different representations were examined, including hash tables, decision trees, and various combinations. The most difficult problem is that while the incoming packets contain specific addresses, the specifications in the configuration file may contain partially specified addresses, and several specifications may match various fields of the packet. Moreover, it is important to provide a deterministic way of selecting between possibly several specifications that all match the packet (the "in order of appearance rule" is arbitrary but easy to comprehend). None of the complicated data structures seemed to have much of an advantage, and all involve complicated implementations.

A simple array of specifications, searched linearly, is easy to implement and provides a deterministic evaluation order. On the other hand, if the configuration file contains many rules, this array may be quite long, and since the individual match evaluations are rather costly, the performance of this approach could be quite bad.

The solution is to combine an array representation of the entire database with a cache recording recent decisions. Since the decisions recorded in the cache are with respect to specific packet headers (actually, only the important fields of these headers), rather than the partial specifications of the configuration file, matching is quite efficient. Since most delay-sensitive applications tend to involve repeated packet exchanges, a small cache with least-recently-used (LRU) replacement should yield high hit rates. (Applications that are sensitive to connection setup time, or that exchange packets infrequently, may suffer low cache hit rates; the delays, however, should be an order of magnitude lower than the propagation delay in a cross-country network. If necessary, specifications pertaining to such delay-sensitive applications can be listed early in the configuration file, to shorten the database search.)

The match function for cache lookups is quite simple. Once the relevant fields of the packet (source and destination addresses, protocol type, and source or destination ports or ICMP type if applicable) are extracted, they are placed into a compact structure that can be compared word-by-word to a cache of previous such records. The cache also includes the decision made for the first occurrence of this pattern. The entries in the cache can never become invalid, since the configuration file does not allow specifications that depend upon any time-varying quantity.

If the cache does not hold a matching record, the entire database must be searched. In this case, the matching function is more complex; it must take into account partial specifications. In particular, each IP address appearing in the packet may have to be converted to a subnet number or network number before comparing with a specification. Since these conversion operations are moderately expensive, they are postponed until necessary, and then done only once per packet. (A subnet number is extracted by first extracting the network number, then using the network number to look up the subnet mask in a hash table built when the configuration file is parsed, and finally applying the network mask to the address in the packet.)

4.3. Supporting fragmented datagrams

One complicating feature of the IP protocol is that if a gateway receives a datagram that is too large to forward in one piece, it may *fragment* the datagram into several packets. Moreover, IP uses "internetwork fragmentation"; the fragments are reassembled only at the destination host, and may not necessarily all flow through any given gateway. Fragmentation is already known to lead to performance problems, potentially severe [14], but it is a necessary evil and IP gateways must forward the fragments they receive.

The problem for the ***screend*** program is that some specifications mention fields, such as TCP or UDP ports, that appear only in the first fragment of a packet. It is thus impossible to decide if subsequent fragments of a TCP or UDP packet should be forwarded without possessing information about the first fragment.

One possible approach would be to simply pass all "non-leading" fragments, those that are not the first fragment of a datagram, regardless of the configuration table. Since normal IP implementations cannot do anything with a received datagram if the first fragment is missing, the action of ***screend*** on the first fragment should have the effect of controlling the entire datagram. Unfortunately, this method leaves open some security holes; for example, two hosts could conspire to exchange information encapsulated in ersatz "fragments", or a malicious host could overwhelm a "victim" host by filling up its fragmentation reassembly buffers.

The approach taken in ***screend*** is to keep a cache of the extracted records for recent "first fragments." When datagram fragments with a non-zero offset (that is, not "first fragments") arrive, the cache is checked to see if we have already seen the corresponding first fragment; if so, the cached information is used. Otherwise, the packet cannot be identified, and is dropped. This solution is not entirely satisfactory, since it requires that all fragments of a datagram flow through one gateway, and that the first fragment appears before all the others. Fortunately, this seems to be true in most cases.

The fragmentation cache is organized as a hash table, with a fixed maximum number of entries; lookups, insertions, and deletions are relatively inexpensive. Since it would require elaborate data structures to detect when we have seen all the fragments of a datagram (and in any case we may never see all the fragments) the hash table must be purged of stale entries whenever it fills up. A stale entry is one for which the ''Time To Live'' field of the corresponding IP datagram has expired; in any event, the lifetime is arbitrarily limited to no more than a few seconds. In order to avoid having to drop packets for lack of space in this table, one must balance the size of the table and the maximum lifetime against the rate of fragment arrival. This is a difficult problem, and may make it impractical to run protocols that extravagantly fragment (such as NFS) over such a gateway.

Fragmentation support complicates the use of a multiprocessor to improve performance, since the fragmentation cache has to be available to all processes if it is to be useful. This means that the cache must be a database shared by all ***screend*** processes, with the associated overhead for locking. Fortunately, this cost is only invoked when fragments arrive.

In general, although ***screend*** does support fragmentation to a certain extent, it is better in almost all cases for source hosts to avoid generating datagrams that will be fragmented. Mechanisms for fragmentation avoidance have been proposed [14, 19], although few have been implemented.

4.4. Performance

The performance of a gateway based on ***screend*** depends on the performance of the underlying gateway implementation, the performance of the kernel gateway screen mechanism, and the performance of the ***screend*** program itself. The latter depends on two major components: the hit rate in the recent-decision cache, and the cost of searching the real database, which in turn depends upon the size and evaluation order of that database. Both the cache hit rate and the database contents are quite specific to a particular installation; the measurements given here only illuminate the extreme cases.

If performance is measured by simply repeatedly sending ICMP Echo messages to a single destination, then the cache hit rate will be 100% (after the first packet), so this case represents the best possible performance. In order to measure worst-case performance, it was necessary to disable the code that entered a decision in the cache, thus forcing a database search on every packet. Additionally, the database was padded with many irrelevant entries, with the actual matching rule coming last. (The ''irrelevant'' entries were constructed to make searching as expensive as possible.)

Table 4-1 shows the measured round-trip and calculated one-way delays for the best case (100% cache hit rate) and for 0% cache hit rates with moderate-sized (10 entry) and large (100 entry) databases. The table also includes some measurements from table 3-1, for comparison; the experimental setup used identical hardware and methodology.

The results for the case of 100% cache hits demonstrate that ***screend*** is quite efficient when its cache is properly sized. Even when the cache is too small, if the number of distinct rules in the configuration file is small, the additional delay (around 0.5 millisecond per packet) is nearly insignificant. The results for the larger configuration database show that it is important to choose both a simple set of rules, and an efficient evaluation order, to avoid noticeable performance degradation. Even in this case, the excess delay is comparable to other delays in a large internetwork.

Varying the packet size, while changing the total round-trip delays, has no significant effect on the incremental delay imposed by the ***screend*** program. This is as expected, since the kernel provides only a prefix of each packet to the ***screend*** program.

The ***screend*** program sources consist of about 2600 lines of commented code. The resulting binary file, compiled for the Vax, is about 48 Kbytes of object code (including library functions), and when running requires on the order of 150 Kbytes of memory, depending on the size of the configuration file.

Time in milliseconds			
Version	Round trip	One way	Added delay
No screen	8	4	
Minimal daemon	10	5	1
screend 100% cache hits	10	5	1
screend 0% cache hits 10 entries	11	5.5	1.5
screend 0% cache hits 100 entries	17	8.5	4.5

Table 4-1: Performance of ***screend*** system

Actual experience with ***screend*** has been quite successful. The system is reliable, easy to configure, and performs well. The ability to log improper packets has made it possible to discover previously obscured problems, including software bugs and "back door" routes that might be security holes.

4.5. Problems and further work

The access control model followed by ***screend*** assumes that application-specific controls (as opposed to host-specific controls) can be based on the TCP or UDP port number of at least one of the end points. This is valid for many TCP and UDP applications, such as remote terminal access or name service, but is not true in all cases. For example, the Berkeley "talk" daemon, used for online conversations between users on different hosts, listens for connection requests on a well-known port number, but builds the actual connection using arbitrary port numbers. The "talk" daemon could be modified to coexist with ***screend***, but this is not always convenient.

Another example where this assumption breaks down, although not directly relevant to the IP-based ***screend***, is found in the "socket" mechanism in the Pup byte-stream protocol (BSP) [1]. Although the initial BSP connection packet is directed at a well-known socket number, the server listening on that socket immediately creates a new socket for the actual connection. Thus, subsequent packets do not travel to a well-known socket number.

Although the well-known port mechanism is followed by existing TCP and UDP applications, it has certain disadvantages related to number management. As a response, it has been proposed that this mechanism be replaced by a "port service multiplexer," a mechanism using port names instead of port numbers [18]. This would make the port-matching support of ***screend*** nearly useless. The Sun RPC system, used with NFS, includes a "port mapper" mechanism that follows the same model [29]. Fortunately, because the RPC packets follow a fixed format, one might extend ***screend*** to parse the RPC headers and filter on the RPC program number.

IP multicasting is another issue not addressed by ***screend***. A standard for IP multicast addressing has only recently been developed [5], and as yet there is little support for or experience with multicasting. Identifying multicast IP addresses is quite simple, so modifying ***screend*** to support the concept of multicast groups should not be difficult.

One attractive extension to ***screend*** would be to use it for datagram accounting. Since the marginal cost of a datagram on a LAN is quite small, datagrams flowing within an organization might be paid for through pooled overhead charges, but some long-distance carriers charge per datagram and an organization might want to charge these costs back to specific sources. The ***screend*** program could easily be extended to record accounting information, for all packets or for those following specific paths. Access control might be used not for security but to restrict long-distance access to those hosts that have promised to pay.

One user has asked that *screend* be enhanced to log the beginning and end of TCP connections (that is, logging packets with the SYN and FIN flags set). This is made difficult by the cache-based implementation of *screend*, because these flags are not part of the packet addressing information. It should be possible to modify the logging mechanism to parse these flags in those packets for which SYN/FIN logging is requested.

Remote network management has recently been an area of active development. Although *screend* is table-driven, it is not difficult to imagine a version that is managed via a protocol such as CMIP [11].

Finally, the current implementation of *screend* does not parse IP header options, which is necessary to defend against use of source routing to avoid simplistic address checks. It would not be difficult to add option parsing, without changing the basic design of the program. Currently, *screend* simply rejects any packets with IP header options.

5. Implementing Visa mechanisms

The implementation described in section 4 uses a configuration file that is stored at the gateway. A *visa* protocol [10] is a different kind of mechanism for implementing datagram access controls. This section sketches the design of visa-protocol support based on the gateway screen mechanism of section 3. This design has not been implemented; an entirely kernel-resident implementation of the visa protocol already exists [9].

5.1. Overview of visa protocols

In a visa protocol, each inter-organization datagram carries an unforgeable mark that proves to a gateway that transmission of the datagram is properly authorized. These marks are called *visas*, by analogy to the stamp made on a passport that allows a bearer to cross a border; visas are unforgeable because they are created using cryptographic mechanisms and secret keys.

In the visa protocols, a gateway has no policy function at all; it simply checks the visa on a packet for validity. Policy decisions are made by a separate ''access control server'' (ACS), which holds a secret key in common with the gateways of its organization. Thus, the principle of policy/mechanism separation is extended even further.

In use, a source host first applies to an ACS for permission to send a packet to a given destination host. The ACS makes a decision based on an arbitrary policy, and it may decide not to issue a visa. If it does issue a visa, there are several alternative protocols that can be followed.

In the first, or ''stateful gateway,'' policy, the ACS creates a new *visa key* to be used for this connection, and transmits the key to both the source host and the appropriate gateways. The source host then creates a new visa for each packet by computing a function based on the packet contents and the key, and attaches the visa to the packet before transmitting it to the gateway for forwarding. The gateway, which has stored its copy of the key in a table, checks to make sure that the visa attached to the packet was computed using the same key.

In the second, or ''stateless gateway,'' policy, the ACS accepts some information from the source host, including a secret session key chosen by the source host. The ACS creates a digitally-signed [7] and encrypted copy of this information, which is returned to the source host. The source host then attaches this visa to each packet, along with a ''signature'' value computed by a function based on the packet contents and the session key, then transmits the packet to a gateway. The gateway decrypts the visa (since it shares the secret encrypt encryption key with the ACS) and extracts the session key; this allows the gateway to validate the visa.

5.2. Implementing visa protocols

The main loop of a visa-protocol daemon is similar to that of *screend*, except that instead of matching the packet header against the configuration database, the daemon must validate the visa contained in the packet header. The algorithms to do this validation are described in detail in [9], and can be treated as ''black boxes'' for the purposes of this paper.

The main difference between the *screend* program and a visa protocol daemon is that *screend* is entirely synchronous, whereas a visa daemon must process events coming from several sources. The main source of events is, of course, packet information arriving via the SIOCSCREEN ioctl, but a visa daemon must also receive and process messages from an ACS. In the stateful-gateway visa protocol, these messages contain the visa keys, and arrive fairly often.

In the stateless-gateway protocol, the messages are used to distribute the secret key shared between the ACS and the gateways; this happens infrequently, when it is determined that the key may have been compromised and a new one is needed.

4.2BSD Unix has several mechanisms for supporting asynchronous event streams. The most elegant is the *select()* system call, which allows a process to wait for an event on any of several I/O streams (file descriptors). The SIOCSCREEN ioctl, however, does not fit this model, since an "event" (the arrival of a new packet) occurs half-way through the request.

The other mechanism is the use of signals (asynchronous software interrupts). A process can request that it receive a signal whenever a message arrives over a specified I/O stream. Since communication with the ACS can be done over such a stream, and because a signal interrupts the execution of the SIOSCREEN ioctl at a "safe" point, the daemon process can accept messages from the ACS while waiting for incoming packets.

It is possible to implement an entirely synchronous visa-protocol daemon by checking for ACS messages whenever the SIOCSCREEN ioctl completes, but before processing the packet. This may lead to extra work in the inner loop, and delays in packet processing, but it is simple to implement. To avoid long delays in ACS-to-gateway transactions, which might cause timeouts, the daemon should use an intermediary process to handle ACS communications.

6. Other Applications

The gateway screen mechanism may be used for other purposes besides access control. For example, the *screend* program could be used solely to log certain kinds of events, such as connections to a service for which clients are charged, or apparently misrouted datagrams.

A similar program might be used to collect statistical information about the flows through a gateway. For example, one might want to know the rate of traffic flowing along a particular set of routes, to support capacity planning. Or, one might want to collect packet traces for use in simulating routing cache policies or packet-header compression algorithms. The gateway screen provides a hook by which a variety of statistics and trace-gathering programs may get access to the complete stream of packet headers.

7. Summary and Conclusions

Gateway-based datagram access controls provide a powerful tool for protecting an organization attached to an internetwork. Ideally, such controls should be incorporated in a dedicated gateway system, but Unix-based gateways are feasible and often fill the need. The kernel-resident gateway screen mechanism described in section 3 provides a simple, flexible, and fairly efficient means of adding access controls to a Unix-based gateway.

The *screend* program described in section 4 implements a flexible, powerful access control policy, without significant performance degradation. Other access control policies, such as the visa mechanism discussed in section 5, should be equally easy to implement in a similar way. Although the performance of a Unix-based gateway may not be equal to a that of a dedicated system, the use of a general purpose operating system together with the gateway screen mechanism makes it easy to experiment with a wide range of access control policies.

8. Acknowledgements

Deborah Estrin got me interested in the issues of datagram access controls; Richard Johnsson, Brian Reid, and Paul Vixie got me interested in devising a specific solution. I also thank Greg Satz and John Shriver for providing information on the capabilities of existing gateway products, Chris Kent for proofreading, and the USENIX referees for pointing out several shortcomings of the original draft.

9. References

[1] David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe. Pup: An Internetwork Architecture. *IEEE Transactions on Communications* COM-28(4):612-624, April, 1980.

[2] *Gateway System Manual.* cisco Systems, Inc., Menlo Park, CA, 1988.

[3] David D. Clark. The Structuring of Systems Using Upcalls. In *Proc. 10th Symposium on Operating Systems Principles*, pages 171-180. Orcas Island, WA, December, 1985.

[4] David D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 106-114. Stanford, CA, August, 1988.

[5] Stephen E. Deering. *Host Extensions for IP Multicasting.* RFC 1054, Network Information Center, SRI International, May, 1988.

[6] —. Blacker Front End Interface Control Document. *DDN Protocol Handbook, Volume 1.* DDN Network Information Center, SRI International, Menlo Park, CA, 1985.

[7] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory* IT-22(11):644-654, November, 1976.

[8] *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications (Version 2.0).* Digital Equipment Corporation, Intel, Xerox, 1982.

[9] Deborah Estrin, Jeffrey C. Mogul, and Gene Tsudik. Visa Protocols for Controlling Inter-Organization Datagram Flow. *IEEE Journal on Selected Areas in Communication* , 1989. In press (Special Issue on Secure Communications).

[10] Deborah Estrin and Gene Tsudik. Visa Scheme for Inter-Organization Network Security. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 174-183. IEEE, April, 1987.

[11] —. Common Management Information Protocol (CMIP). ISO 9595/2.

[12] Van Jacobson. Congestion Avoidance and Control. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 314-329. Stanford, CA, August, 1988.

[13] S. C. Johnson. *Yacc -- Yet Another Compiler-Compiler.* Technical Report 32, Bell Laboratories, Murray Hill, New Jersey, July, 1975.

[14] Christopher A. Kent and Jeffrey C. Mogul. Fragmentation Considered Harmful. In *Proc. SIGCOMM '87 Workshop on Frontiers in Computer Communications Technology*, pages 390-401. Stowe, VT, August, 1987.

[15] B. Leiner, J. Postel, R. Cole, and D. Mills. The DARPA Internet protocol suite. In *Proc. INFOCOMM 85.* Washington, DC, March, 1985. Also in *IEEE Communications Magazine*, March, 1985.

[16] M. E. Lesk. *Lex -- A Lexical Analyzer Generator.* Technical Report 39, Bell Laboratories, Murray Hill, New Jersey, October, 1975.

[17] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proc. 5th Symposium on Operating Systems Principles*, pages 132-140. ACM, November, 1975.

[18] Mark Lottor. *TCP Port Service Multiplexer (TCPMUX).* RFC 1078, Network Information Center, SRI International, November, 1988.

[19] Jeffrey C. Mogul, Christopher A. Kent, Craig Partridge, and Keith McCloghrie. *IP MTU Discovery Options.* RFC 1063, Network Information Center, SRI International, July, 1988.

[20] Jeffrey Mogul and Jon Postel. *Internet Standard Subnetting Procedure.* RFC 950, Network Information Center, SRI International, August, 1985.

[21] Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proc. 11th Symposium on Operating Systems Principles*, pages 39-51. Austin, Texas, November, 1987.

[22] John Nagle. On Packet Switches With Infinite Storage. *IEEE Transactions on Communications* COM-35(4):435-438, April, 1987.

[23] Jon Postel. *User Datagram Protocol.* RFC 768, Network Information Center, SRI International, August, 1980.

[24] Jon Postel. *Internet Protocol.* RFC 791, Network Information Center, SRI International, September, 1981.

[25] Jon Postel. *Transmission Control Protocol.* RFC 793, Network Information Center, SRI International, September, 1981.

[26] Jon Postel. *Internet Control Message Protocol.* RFC 792, Network Information Center, SRI International, September, 1981.

[27] Proteon, Inc. *ProNET p4200 Gateway Software User's Guide.* Westboro, MA, 1988.

[28] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conference*, pages 119-130. Portland, OR, June, 1985.

[29] *Network Programming.* Sun Microsystems, Inc., Mountain View, CA, 1988.

[30] Hubert Zimmermann. OSI Reference Model -- The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications* COM-28:425-432, April, 1980.

Appendix I. Grammar for the screend configuration file

This is an informal guide to the grammar of the *screend* configuration file. It is meant for readers who are familiar with the basic concepts of the IP protocol family.

I.1. Lexical structure

- **Comments** can either be "C-style" comments, delimited by "/*" and "*/", or "csh-style" comments begun with "#" and terminated by the end of a line. Comments do not nest.
- **Case** is significant in reserved words (all are lower-case). This is actually a benefit, because if a host name happens to conflict with a reserved word, you can use the host name in upper-case.
- **Host names** begin with alphabetics but may contain digits, '-', '.', and '_'. The same is true of network, subnet, and netmask names. All can also be entered in dotted quad notation (for example, "10.1.0.11").
- **Numbers** may be in decimal or in hex (0x0 notation). Octal notation is not allowed because nobody uses it in this context. (Actually, hex is almost as useless).
- **Protocol names** and **port names** (for TCP or UDP) are as in `/etc/protocols` and `/etc/services`, respectively. These can also be given as numbers (host byte order).
- **ICMP type codes** must be chosen from this list, or given as numbers:

```
echoreply                       timeexceeded
unreachable                     parameterproblem
sourcequench
redirect
echo
```

```
timestamp
timestampreply
informationrequest
informationrreply
addressmaskrequest
addressmaskreply
```

- **All white space** is the same (including newlines).

I.2. Syntax

General syntax rules:

1. The configuration file consists of ''specifications'' terminated by semicolons.
2. There are three kinds of specifications:
   a. **default action specification**: There should only be one of these (the last one is the one that counts); it specifies what action to take if no action specification matches a packet.
   b. **subnet mask specifications**: specifies the subnet mask used for a given network.
   c. **action specifications**: specifies a class of packets and the action to take when such a packet is received.
3. Specifications can appear in any order, but the evaluation order of action specifications is the order in which they appear in the file.

In BNF, this is:

<configuration-file> ::= { *<specification>* | *<configuration-file>* *<specification>* }
<specification> ::= { *<default-action>* | *<subnet-spec>* | *<action-spec>* }

The syntax for a default action specification is:

<default-action> ::= **default** {**accept** | **reject**} [**notify**] [**log**] **;**

Note that ''default accept notify;'' is legal but the ''notify'' in this case is a no-op. If not specified, the default action is ''reject''.

The syntax for subnet mask specifications is:

<subnet-spec> ::= **for** *<network>* **netmask is** *<maskval>* **;**

The *<network>* is either a network name or a dotted-quad address, such as ''36.0.0.0''. ''36'' is *not* a reasonable value. *<Maskval>* is either a name (treated as a hostname) or a dotted-quad address, such as ''255.255.255.0'' (bits are *on* for the network and subnet fields.)

The syntax for action specifications is:

<action-spec> ::= **from** *<object>* **to** *<object>* {**accept** | **reject**} [**notify**] [**log**] **;**

Such a specification says that packets flowing this way between this pair of ''objects'' (defined below) should either be accepted or rejected. If ''notify'' is specified, when a packet is rejected an ICMP error message is returned to the source. If ''log'' is specified, this packet and its disposition are logged.

Conceptually, for each packet the action specifications are searched in the order they appear in the configuration file, until one matches. The specified action is then performed. If no specification matches, the default action is performed.

To simplify the configuration file, the syntax

<action-spec> ::= **between** *<object>* **and** *<object>* {**accept** | **reject**} [**notify**] [**log**] **;**

may be used to indicate that the same action should be performed on packets flowing in either direction between the specified pair of ''objects.'' Note that this is simply syntactic sugar; it has the same effect as specifying the two unidirectional rules, with the ''forward'' direction listed first.

An ''object'' is a specification of the source or destination of a packet. The syntax for object specifications is somewhat complex, since certain fields are optional:

<object> ::= { *<address-spec>* | *<port-spec>* | *<address-spec> <port-spec>* }

If the *<address-spec>* is not given, "any host" is assumed. If the *<port-spec>* is not given, "any protocol and port" is assumed.

<address-spec> ::= { *<net-spec>* | *<subnet-spec>* | *<host-spec>* | **any** }

<net-spec> ::= { **net** *<name-or-addr>* | **net-not** *<name-or-addr>* }
<subnet-spec> ::= { **subnet** *<name-or-addr>* | **subnet-not** *<name-or-addr>* }
<host-spec> ::= { **host** *<name-or-addr>* | **host-not** *<name-or-addr>* }

The "-not" convention means that the object specification matches if the specified field does *not* have the specified value. For example, "`from host-not sri-nic.arpa to host any reject`" means that packets *not* from `sri-nic.arpa` are dropped. The "subnet" and "subnet-not" forms match against the entire address under the subnet mask (for example, if the netmask for net 36 is 255.255.0.0, then "`subnet 36.8.0.0`" matches a packet address of 36.8.0.1).

<name-or-addr> ::= { *<name>* | *<dotted-quad>* | **any** }

<port-spec> ::= { **proto** *<proto-name-or-number>*
| **icmp type** *<type-name-or-number>* | **icmp type-not** *<type-name-or-number>*
| **tcp port** *<port-name-or-number>* | **tcp port-not** *<port-name-or-number>*
| **udp port** *<port-name-or-number>* | **udp port-not** *<port-name-or-number>* }

<proto-name-or-number> ::= { *<name>* | *<number>* }
<type-name-or-number> ::= { *<name>* | *<number>* | **any** | **infotype** }
<port-name-or-number> ::= { *<name>* | *<number>* | **any** | **reserved** }

"Reserved" ports are those reserved by 4.2BSD Unix for privileged processes. "Infotype" ICMP packets are those that are purely "informational": echo, timestamp, information, and addressmask requests, and the corresponding replies.

Security Testing of AIX System Calls Using Prolog

Janet A. Cugini, Shau-Ping Lo, Matthew S. Hecht, Chii-Ren Tsai†*,*
Virgil D. Gligor‡*, Radhakrishna Aditham, T. John Wei*

IBM Systems Integration Division
708 Quince Orchard Road
Gaithersburg, Maryland 20878

ABSTRACT

We wrote security test plans and, using Prolog, an automated security testing system for C2 security testing of AIX[1] Version 2.2.1 system calls [TCSEC 85]. AIX 2.2.1 is a version of the UNIX[2] operating system that runs on the IBM RT PC hardware and that contains security features (for the base operating system and TCP/IP) *designed to satisfy* the C2 security requirements and the B3 trusted path requirement of [TCSEC 85]. The notable design features of our work include: (1) it is based on the "gray box" security testing method of [Gligor 87b]; (2) the security test plans contain both *coverage analysis*, which explains why the test conditions are sufficient and why the test data cover the test conditions, and system call test environment *dependency analysis*; (3) the security testing system is *comprehensive* in that it automates all aspects of security testing except for writing security test plans and adding, per system call, a set of facts and rules that cover system call syntax, test conditions, and expected outcomes; and (4) this work is also suitable for security testing of UNIX system calls for higher security classes [TCSEC 85]. We explain what motivated the Prolog security testing system approach, discuss how we write security test plans, sketch the structure of the security testing system, and provide quantitative experience with this approach to security testing. From this experience and the Secure XENIX[3] project experience, we conclude that comprehensive security testing of UNIX system calls with an automated security testing system in Prolog is preferable to writing and maintaining security test programs manually written in the C programming language.

Key Words and Phrases:

security, testing, security testing, operating systems, UNIX operating system, UNIX system calls, operating system security, computing security, Prolog, logic programming.

1. Introduction

Problem Statement. Our experience with Secure XENIX [Gligor 87a, Gligor 87b, Hecht 87] suggests that Security Testing is *the* most expensive component at security level B2 in the development and evaluation of a trusted computer operating system or of a trusted application (e.g., communication subsystem, relational database system). Thus, we **need to rethink how to do Security Testing to decrease the cost significantly** beyond the normal leverage of having previous experience.

† This author's address is: VDG, Inc., 4901 Derussey Parkway, Chevy Chase, MD 20815.

‡ This author's address is: Electrical Engineering Dept., Univ. of Maryland, College Park, MD 20742.

[1] AIX, RT, RT PC, and RT Personal Computer are trademarks of International Business Machines Corporation.

[2] UNIX is a registered trademark of AT&T Bell Laboratories.

[3] XENIX is a registered trademark of Microsoft Corporation. *Secure XENIX* is an IBM Systems Integration Division product offering (available from July 1987).

For the Secure XENIX project, targeted for B2 security (with B3 features but only B2 assurance), we produced about 4,300 net pages of security evaluation supporting evidence with about 19 PYs (person–years) of effort, over three calendar years; this does not count design and implementation effort for security features, only the paper evidence. This effort, which dominates the code development effort for security features, represents about six persons full time for three years to produce the equivalent of over ten 400–page books. Dividing 4,300 pages by 19 PYs, our productivity was about one page of documentation per person per working day, where there are five working days per week. The largest component of this cost was Security Testing and Security Testing Documentation, at about 102 PMs (person–months), which is 8.5 PYs, and about 1,900 pages, with the same productivity figure. The largest component of Security Testing was producing the Kernel Security Test Plan (1,339 pages, 18 PMs) and Kernel Security Test Programs (C code, 36 PMs).

About Our Solution. To address the problem of a time–consuming, potentially error–prone process of manually writing and maintaining C test programs for security test plans and manually establishing the test plan–program correspondence, we decided to use a high–level declarative language, Prolog, for writing directly executable test scripts, thus eliminating the step of manually writing (revising, maintaining) C test programs and simplifying the test plan–program correspondence step. We selected Quintus Prolog [Quintus 86] because it is available for AIX/RT as a compiler, offering speed over an interpreter, and because one can link edit C program modules with Prolog modules; in other words, one can invoke AIX system calls from a Prolog program if desired. In reviewing the entire Security Testing process, we observed that the first step of writing test *plans* with coverage analysis remains a human activity, but that the second step of manually writing test *scripts* or *programs*, and assuring correspondence, and writing test results reports can be improved significantly in automation and thus cost.

This was *not* a simple matter of "porting" the existing plan from Secure XENIX, by projecting out some B2–specific (e.g., MAC and ACLs [Gligor 87a]) and XENIX–specific items then adding AIX–specific items, and "porting" the existing test programs in C. We made this decision because there was significant work for AIX 2.2.1 over and above any document and code porting activity, and because our calculations projected that it would take less time and cost to redo the entire testing effort in Prolog and write new scripts in Prolog than to revise the existing ones and write AIX–specific ones.

Prolog is a declarative language. A Prolog programmer writes facts and rules of inference, rather than low level procedural code. A Prolog program user asks questions involving the facts and rules. Prolog comes with a software "inference engine" (mechanism) built into it, and a backtracking mechanism. A Prolog program consists of facts and rules and questions. To answer a question, Prolog uses the supplied facts and rules, the inference engine, and backtracking. The backtracking mechanism is well–suited to testing in which one needs to walk through an array of expected test outcomes.

For this project, the fact and rule sets for system calls contain 2,310 facts and rules, and the library (of common "procedures") and test driver (to run the tests) contains 166 and 103, respectively, for a total of 269, procedures or rules. These rules are remarkable in the sense that they create and afterwards delete the testing environment objects, write C programs, compile them, run them, collect the results, compare the actual results to predicted results, and write a results report.

Our solution to Security Testing of AIX system calls has the following advantages over previous work in this area. First, it combines "gray box" Security Testing [Gligor 87b] of AIX system calls with a security testing system written in Prolog, together resulting in a *succinct representation* of nonredundant tests in a declarative language. Second, it offers *more automation* of the Security Testing process. Third, it is *easily extensible* to new system calls, to changes in test conditions, and to higher security classes.

Extant Work. We know of four studies that pertain to this work. The "gray box" Security Testing work on the Secure XENIX [Gligor 87a, Gligor 87b, Hecht 87] kernel is the foundation for our work, but it neither used nor proposed an automated security testing system implementation. Pesch and others [Pesch 85] used a knowledge–based technique and Prolog to automate the functional testing of about 60 UNIX system calls, but they did not address Security Testing. They designed a test specification language, and wrote a compiler for it in Prolog to translate a test specification into a C test program. In contrast, we use Prolog as a *testing engine*, not a compiler. Pesch describes a prototype system, and uses future verb tense to describe some parts of it; unlike our work, they provide no quantitative experience. For Secure XENIX, IBM SID developed a Security Testing process with test plans (excluding test environment dependency analysis) written in English and test programs manually written in C. A "test operator" program simulating a human operator on a "vanilla" XENIX system runs this code on a target system. Leventer and Prevost [Leventer 87] sketched a non-procedural approach using Prolog for Security Testing as a feasibility experiment and precursor to our work. In general, Security Testing experience is conspicuous by its absence in the software testing literature.

Audience Assumptions. We assume that the reader is somewhat familiar with: the AIX (or UNIX) system calls and commands (e.g., [AIX TR] and [AIX CR]); the *Orange Book* requirements for C2 [TCSEC 85]; the "gray box" testing method [Gligor 87b]; and, Prolog (e.g., [Quintus 86]).

Structure of This Paper. The rest of this paper has five sections. Section 2 summarizes the requirements, goals, and practices of Security Testing. Section 3 briefly reviews the writing of security test plans for AIX system calls. Section 4 sketches the design of our automated security testing system. Section 5 shares some quantitative experience. In Section 6 we draw our conclusions.

2. Requirements, Goals, and Practices

Security Testing has requirements over and above general testing. To understand this, we first differentiate Security Testing from Functional Testing. Next, we state and distinguish our requirements (what we must do), goals (what we strive to do), and practices (what we actually do).

2.1. Security Testing versus Functional Testing

Security Testing is based on the *Orange Book* [TCSEC 85] requirements. This means that we are done when we satisfy a specific list of security requirements. Also, it means that we must define a *security model* (set of security assumptions that define what is security relevant) and the *Trusted Computing Base*, or *TCB* (see [TCSEC 85]), including the TCB components, TCB interface, classes of TCB subjects, classes of TCB objects, and the security policy. The security policy model has access control (discretionary, mandatory, object reuse, labeling) and accountability (identification and authentication, trusted path, auditing) components. Here we focus on the access control component; similar reasoning applies to the accountability model. While a security model is not explicitly required in [TCSEC 85] until class B1, it is implicitly required at class C2 to help define "security relevant" for auditing and Security Testing. Security model definition and TCB identification, both outside the scope of this paper, must precede Security Testing to understand what is security relevant. In this work we define a system call as *security relevant* if it reads or writes data described in the security model; this is in contrast to the specious definition of "it changes the security state." The AIX kernel is only one component of the AIX/RT TCB, where its interface is the set of AIX system calls. Security Testing of other AIX/RT TCB components is outside the scope of this paper.

Security Testing, which covers all and only security relevant system correctness assertions, is a subset of *Functional Testing*, which covers all system correctness assertions. Also, Security Testing focuses only on the use of functions by a user (or client) at the TCB interface; among other things, Functional Testing covers the use of functions at the TCB interface by a privileged user. Our focus here is Security Testing, not Functional Testing.

It is commonly believed that "system testing" of operating system products that include security features is thorough, and that it also covers Security Testing. Without convincing evidence such as test plans with coverage analysis, typically Functional Testing does not already subsume Security Testing. For example, a previous AIX functional test for system call **access** invoked it six (6) times, whereas our security test plan for access identifies 1,320 necessary invocations. While it is possible to "beef up" Functional Testing to include Security Testing, it is more desirable to keep Security Testing separate because of its additional requirements.

2.2. General Testing Requirements

For modular and incremental testing and simplified partial retesting, it is understood, as an implicit requirement, that *the testing granularity is a single system call.* In other words, if so specified, any single system call can be tested alone, any subset of system calls can be tested, and any list of system calls, possibly with repetitions, can be tested. This implies that, to test a system call, we (1) clean the test environment, (2) create the test environment, and (3) run the test. Performing both steps (1) and (2), before (3), guarantees that we obliviously establish test environment preconditions.

Coverage analysis, dependency analysis, and repeatability are fundamental to testing.

Requirement 1. Test Plan with Coverage Analysis. Prior to writing and running tests, write a test plan that identifies the test conditions, the test data, and the expected outcomes. The test plan contains coverage analysis that explains why the test conditions are sufficient and why the test data cover all the test conditions.

Requirement 2. Test Dependency Analysis and Test Dependency Reduction. Identify test environment dependencies and test program dependencies, analyze the dependencies, and remove as many dependencies as is practical.

The test plan should identify any testing dependencies known when the test plan is written. A *test dependency* is a *correctness dependency*. We say that a *test dependency* exists between two programs P and Q, or P *depends on* Q, if the correctness of P depends on the correctness of Q. This "depends on" relation is transitive. Removing test dependencies is desirable when it simplifies (or speeds up) the correctness reasoning, writing, running, or maintaining of test programs.

The *test environment* for a system call test program consists of all the things that must be created and present to test the system call: user accounts, groups, and test objects (files, directories, ...) with their specified properties (e.g., access privileges). A *test environment dependency* exists between the test program for system call A and the test program for system call B if the test program for A is executed before the test program for B, and the test environment for B shares part or all of the previous test environment of A. A test environment dependency is undesirable because it violates our implicit requirement for unitary testing granularity. To minimize test environment dependencies, test environments should be reinitialized anew for different system call tests. This implies that the number of different test objects created and logins executed may become very large, which implies that Security Testing should be automated as much as possible.

A *test program dependency* exists between a test program for system call A and a test program for system call B if the test program for A (directly or indirectly through a sequence of calls) invokes B. A *cyclic test dependency* exists between a test program for system call A and a test program for system call B if the test program for A invokes B, and the test program for B invokes A. A cyclic test dependency is undesirable because the correctness reasoning for a test program is not as simple as it is otherwise. Test program dependencies and cyclic test dependencies should be eliminated whenever possible.

Requirement 3. Test Results Repeatability. All tests must comply with the test plan, be complete (i.e., all test conditions covered), and be repeatable. All results must be documented, with predicted and actual outcomes compared. Since the test plan contains (fixed) expected test outcomes, tests must be repeatable for different groups (e.g., developers and evaluators) to compare actual to expected outcomes.

2.3. Security Testing Requirements

The TCB is the principal operating system component that implements security mechanisms and policies, and the TCB itself must be protected. TCB protection is provided by a "reference monitor" concept whose data structures and code are isolated, noncircumventable, and small enough to be verifiable.

Requirement 4. Only External Testing at TCB Interface. To preserve TCB isolation and noncircumventability, conduct Security Testing only from outside the TCB, at the TCB interface.

Three significant consequences follow from this. First, *no TCB instrumentation* is allowed. New data structures and code should not be added to the TCB for Security Testing purposes, and special TCB entry points that are unavailable to normal user programs should not be used. Second, *no special operation mode* should be used for Security Testing. This means that only unprivileged test programs, which do not directly read or write internal TCB data structures or code, are allowed. Also, Security Testing should be done for a TCB configured and installed to operate in normal mode (not maintenance mode and not a special test mode). Third, it *may be impossible to eliminate some cyclic test dependencies* without TCB instrumentation.

Requirement 5. Security Test Condition Selection, Test Data Selection, Coverage Analysis. For the Security Testing of system calls, derive security test conditions both from an interpretation of the security model and from the specification of individual system calls. For security model dependent coverage, security test data includes system call input data, return code, TCB object type, TCB object hierarchy, TCB subject identification, access privilege, security levels of each subject and object, authorization–check coverage, and so on. For individual system call coverage, security test data selection includes boundary–value analysis [Gligor 87b, NCSC 88] and, for "access graph dependencies" [Gligor 87b], data–flow analysis showing that every node and arc of the access graph is traversed in proper sequence.

2.4. General Testing Goals

Two general testing goals are (1) automation and (2) nonredundant testing, the elimination of redundant tests without loss of coverage.

2.5. Security Testing Practices

Since it is typically impractical or impossible to resolve all cyclic test dependencies, we must make engineering judgments to proceed with the art of Security Testing. Below we discuss such practices.

On Using Untrusted Testing Tools. One can use untrusted tools to test the trust of an operating system. The whole issue of Security Testing raises questions on the trust of the tools, such as Quintus Prolog, being used for such testing. In general, it is not necessary to use trusted tools to test the security of an operating system. It is enough to show that the tools being used do not circumvent, violate, or supersede any security policy. *Some degree of this assurance can be obtained by turning on auditing while performing the tests.* In fact, there is no practical way of avoiding the use of untrusted tools for Security Testing. For example, the C compiler is untrusted, and the programs written to test a particular system call must use this compiler. This is true regardless of whether the test programs are created and compiled by an individual or through the use of a tool such as Quintus Prolog. Therefore, using untrusted tools for Security Testing is justifiable and unavoidable.

On Testing Order and the Inevitability of Cyclic Test Dependencies. If some system call test fails, then it is necessary to retest everything that is dependent on that system call since other cases may be in error. Our experience suggests that there is no practical way of breaking all cyclic dependencies among AIX (and UNIX) system calls. At best, *designing and running a suite of secondary tests involving other cyclic dependencies can provide more assurance*, but we did this neither for Secure XENIX nor for AIX 2.2.1. To minimize retesting work when a test fails, testing order should be: independent, access graph dependent, access check dependent (discussed below).

On Using an Operating System To Help Test Itself. One can do Security Testing of an operating system on the same operating system. Assume that we run non–security–relevant functional tests for system calls first, then we run security tests, and assume that all tests succeed. A security test program for a system call ultimately depends on the already established functional correctness of **fork** and **exec** to run the test program completely as is without modification. Use of unprivileged tools (e.g., **sh**) to run test programs is not a problem since we can use auditing to verify system call executions. Thus, performing Security Testing on one RT suffices. Our testing system can be configured for one RT or two RTs (to minimize Quintus Prolog licenses). When configured for two RTs, we use TCP/IP, a part of the AIX 2.2.1 TCB [Hecht 88, Tsai 89], and we use Prolog only on the tester (not target) RT.

On Test Environment Dependency Reduction. Test one system call at a time recreating anew the testing environment objects for each system call. In practice, for AIX 2.2.1, we create all *subject* information (users, groups) only once.

3. Writing Security Test Plans for AIX System Calls

This section explains how we identify security test conditions for AIX system calls, and how we determine their sufficiency. See [Gligor 87b] for a more complete explanation.

The purpose of Security Testing is to gain assurance that a system operates within the constraints of a given set of policies and mechanisms. The security policies and mechanisms to be tested for AIX 2.2.1 system calls are:

Discretionary Access Control (DAC),
Object Reuse (OR), and
Trusted Computing Base (TCB) Isolation.

The DAC policy enforces the access control between named users and named objects in the system. Specifically, the DAC policy requires that all functions which relate to access control in the system calls be tested. The OR policy requires that when a storage object is initially assigned, allocated, or reallocated to a subject from the TCB's pool of unused storage objects, the TCB shall assure that the object contains no data for which the subject has no authorization. The OR policy should be tested against all storage objects in AIX. Since the system call layer is the lowest layer that has the ordinary user interface for creating the AIX storage objects, we include object reuse testing in system call testing. The TCB Isolation mechanism allows the TCB to maintain a domain for its own execution that protects it from external interference or tampering. The testing of the TCB Isolation mechanism includes *process address space isolation* and *kernel isolation*.

3.1. System Call Security Testing Method

The gray box testing method eliminates redundant tests through systematic test selection and coverage analysis. To reduce redundant tests, it uses the access check graph of the kernel, the call graph of kernel functions that check access to TCB objects, and the analysis of access check dependencies.

Applying the gray box testing method to the AIX 2.2.1 system calls security testing, we categorize the test for system calls into the following testing types:

(1) Not Security Relevant,
(2) Independent,
(3) Access Graph Dependent,
(4) Access Check Dependent,
(5) Privileged,
(6) Process Address Space Isolation, and
(7) Kernel Isolation.

The test of a system call may fall into one or more testing types; Figure 1 lists the AIX 2.2.1 system calls under different testing types, and for these system calls Figure 2 shows the access graph dependencies (AGDs) and access check dependencies (ACDs). Observe in Figure 2 that AGD system calls have a path name argument, and ACD system calls have an *object handle* argument like a file descriptor, obtained from an *object open* or *object create* system call.

Not Security Relevant. In general, a system call is *not security relevant* if nothing outside the process's domain is affected by the execution of the call. No test needs to be done for a non-security relevant system call.

Independent. The testing type of a system call is *independent* if the testing of this system call does not rely on the testing of any other system call. For an independent system call, we thus need to do exhaustive testing for all the security relevant functions of the system call. We identify a system call as an independent system call if (1) this system call does not share any security relevant code (e.g., code for doing an access check) with other system calls in the kernel, or (2) this system call does share security relevant code with other system calls and the security testing of this system call covers the most part of the testing for the shared codes among all system calls that share those codes.

Testing Type	#	System Call Names of This Testing Type
1. Not Security Relevant	27	alarm, close, dup, exit, getdomainname, getegid, geteuid, getgid, getgroups, gethostid, getpgrp, gethostname, getpid, getppid, getuid, kleenup, nice, pause, sbrk, setpgrp, sigblock, sigpause, sigsetmas, sync, umask, wait, wait3.
2. Independent	6	access, fork, kill, msgget, semget, shmget.
3. Access Graph Dependent	25	chdir, chmod, chown, chownx, creat, execve, fullstat, link, lstat, mkdir, mknod, open, openx, readlink, rename, revoke, rmdir, stat, statfs, symlink, tcb, unlink, utime, uvmount, vmount.
4. Access Check Dependent	28	fchmod, fchown, fclear, fcntl, ffullstat, frevoke, fstat, fstatfs, fsync, ftruncate, getdirentries, ioctl, ioctlx, lockf, lseek, msgctl, msgrcv, msgsnd, msgxrcv, read, readx, semctl, semop, shmat, shmctl, shmdt, write, writex.
5. Privileged	39	acct, audit, auditbin, auditevents, auditlog, auditproc, chmod, chown, chownx, chroot, fchmod, fchown, frevoke, iplvm, link, loadtbl, mknod, mount, msgctl, plock, reboot, revoke, setdomainname, setgid, setgroups, sethostid, sethostname, setreuid, setregid, setuid, stime, symlink, tcb, ulimit, umount, unlink, uvmount, vmount, waitvm.
6. Process Address Space Isolation	57	brk, chdir, chmod, chown, chownx, creat, disclaim, execve, ffullstat, fstat, fstatfs, fullstat, getdirentries, ioctl, ioctlx, link, lstat, mkdir, mknod, mntctl, msgctl, msgrcv, msgsnd, msgxrcv, open, openx, pipe, profil, ptrace, read, readlink, readx, rename, revoke, rmdir, select, semctl, semop, shmat, shmctl, signal, sigstack, sigvec, stat, statfs, symlink, tcb, time, uname, unamex, unlink, usrinfo, utime, uvmount, vmount, write, writex.
7. Object Reuse	8	creat, open, mkdir, mknod, pipe, msgget, semget, shmget

Figure 1. AIX 2.2.1 System Calls Catagorized by Security Testing Type

Access Graph Dependent. An access check graph is a graph in which each node represents an access check function (subroutine, procedure) and each arc identifies the data flow from one access check function to the next access check function. Typical inputs to an access check function may consist of object identities, object types, and required access privileges, whereas the typical output may consist of the input to the next function (as defined above), or the outcome of the function check. Sequencing information for access check functions embedded in the access check graph consists of (1) the ordering of these functions, and (2) the number of arc traversals for each arc. To save space in this paper, we omit the access check graph for the AIX 2.2.1 system calls.

This sequencing information helps eliminate a significant number of test conditions and associated tests for the system calls shown in the access check graph of the system calls. This is because, in principle, each system call may have a different access check graph; whereas *in practice, substantial parts of the graphs overlap.* Consequently, if one of the graph paths is tested with sufficient coverage for a system call, then the test conditions generated for a different system call whose graph overlaps with the first system call need only include the access checks specific to the latter system call. This is true because, by the definition of the access check graph, the *commonality of paths* means that the same access checks are performed in the same sequence, on the same types of objects, and with the same outcomes (i.e., success and failure returns). However, the specific access checks of a system call must also show that the untested subpath(s) of the system call join(s) the tested path. We perform this path join test simply by showing that, for each type of object used in the access check path for the latter system call, the test result is consistent with the result obtained for the first (i.e., independent) system call. For example, the discretionary access check for the system calls **access**, **open**, and **creat** is actually performed by the same kernel private function "access" as shown in the (omitted) access check graph. Therefore, if we already did exhaustive testing for the **access** system call, then we do not need to repeat the testing for those access check tests performed by the "access" function that are common with those of the **open** or **creat** system call. The only test we need to do is to choose one access check test for **open** or **creat** and show that the test result is consistent with the test result obtained from **access** system call testing.

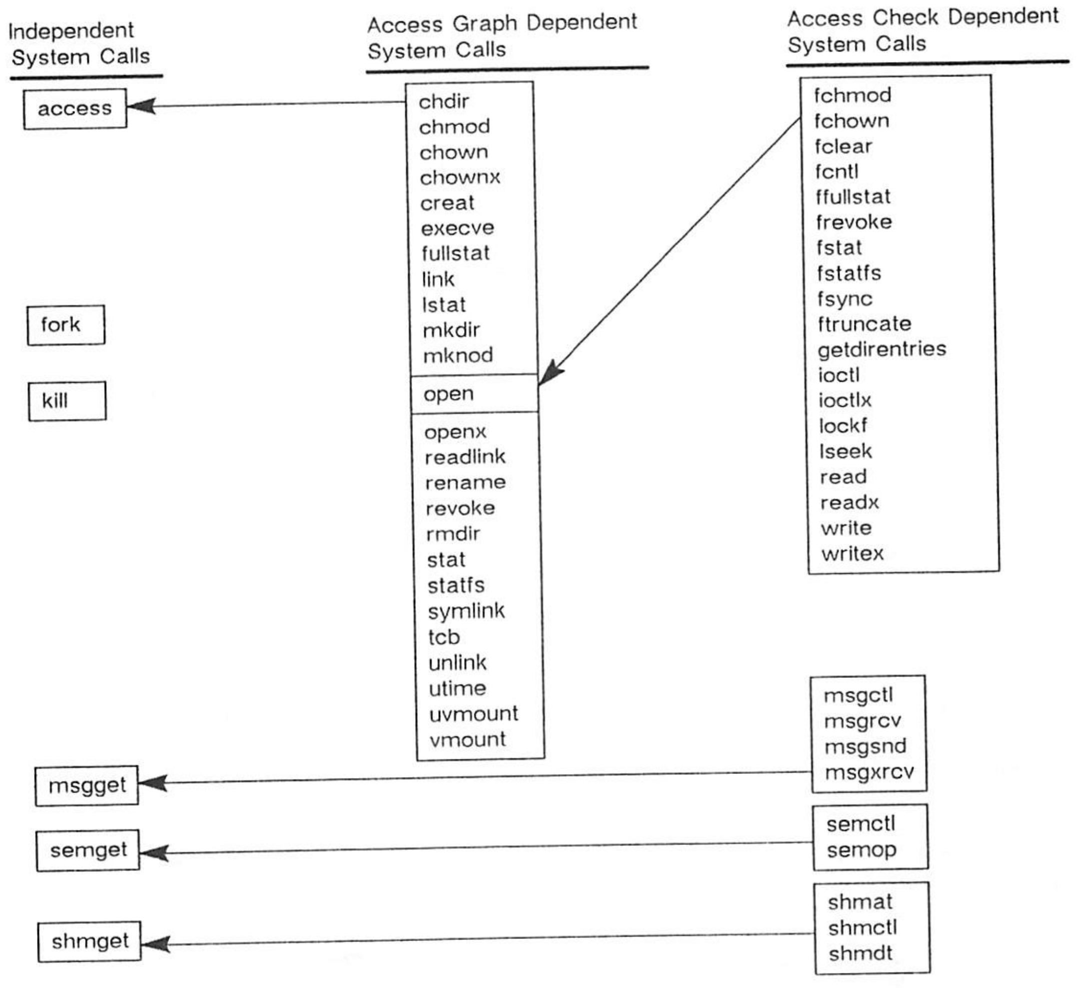


Figure 2. Access Graph Dependencies and Access Check Dependencies of AIX 2.2.1 System Calls.

Access Check Dependent. The security testing of a system call A is *access check dependent* on the specification of a system call B if a subset of the access checks needed in system call A are performed in system call B, and if system call B must always precede system call A (i.e., system call A fails if system call B has not been done or has not been successfully completed). In the case of such dependencies, it is sufficient to test system call B first and then to test only the access checks of system call A that are not performed in system call B (if any). However, the existence of the access check dependency must still be tested. For example, in the case of the **write** system call, its specification is dependent on the discretionary access checks of the **open** call.

Therefore, to test the **write** system call, we first test the existence of the access check dependency of **write** on **open**. We then perform additional tests not done in **open**. In this case, one example of an additional test is to show that **write** will fail for writing beyond the file size limit.

Privileged. A *privileged system call* or *privileged option* can be performed only by the superuser on AIX 2.2.1. For each such privileged system call, we need to demonstrate that an unprivileged user cannot enter the TCB to perform the function specified by the privileged system call through the privileged call entry. This is a part of the TCB Isolation testing. However, some of the privileged system calls grant the execution permission for the call to a limited number of unprivileged users. For example, the **chown** system call allows only the owner of the object or the superuser to invoke the call on a given object. In this case, we need to demonstrate that the system call returns failure whenever a normal user who is not the owner of the object invokes the system call.

Process Address Space Isolation. A system call needs *process address space isolation* testing, or parameter validation, if it uses one or more address parameters as input parameters. All parameters passed by address to the kernel are validated by the kernel before use. That is, a system call cannot supply to the kernel addresses that are outside of the invoking program's process address space. This test is also part of the TCB Isolation testing. The test condition for process address space isolation is the same for all system calls that need this test.

Object Reuse. For object reuse testing, we use an exhaustive testing approach; all storage objects that can be created, allocated, or reallocated through the unprivileged user interface are tested against the object reuse requirement. AIX 2.2.1 TCB object classes include files, directories, devices, named pipes, message queues, semaphores, and shared memory segments.

Kernel Isolation. Kernel isolation testing must show that there is only one valid entry into the kernel; this can be achieved thought boundary value coverage of all possible "supervisor calls (SVCs)" or simply through code review. For AIX 2.2.1, we did this by code reviews.

3.2. Order of Writing Security Test Plans

The order of writing the system call test plans is important due to the introduction of the access graph dependencies and access check dependencies of system calls through the gray box testing method. To be consistent with our testing method, we developed the system call test plans in the following order:
(1) the not security relevant system calls,
(2) the access check graph for AIX 2.2.1 system calls,
(3) the independent system calls,
(4) the system calls with access graph dependencies,
(5) the system calls on which other system calls are access graph dependent,
(6) the system calls with access check dependencies, and
(7) object reuse, process address space isolation, and kernel isolation.

3.3. Test Plan Format

We use a common outline for the security test plan of each system call. Figure 3 shows an example of this format for the **access** system call. The general outline items are: Introduction, Test Condition Selection, Test Data Selection, Coverage Analysis, and Test Environment Dependencies. Figure 4 contains the security test plan for **stat**. Reference [NCSC 88] recommends a similar format, but not containing the first and last items.

4. Design Sketch of the Security Testing System

Our Security Testing system, which the first author wrote in Prolog, has three major parts: (1) the fact and rule sets for system calls, (2) the test driver, and (3) a library of general Prolog predicates. A complete description of this system includes its overall architecture, the list of Prolog predicates, a definition with rationale with examples of each Prolog predicate, how to read and write a fact and rule set from a test plan for a system call, how the system tests a system call (i.e., "executes" the fact and rule set for a system call), how the system writes the results report, how to configure the system, and how to run the system. Since a complete testing system description is better suited to a testing system user's manual, which we wrote too, and would far exceed the length target of this paper, here we only sketch parts of the fact and rule set and test driver designs.

1. Introduction

2. Test Condition Selection

3. Test Data Selection
   - 3.1 Data for Condition 1: Discretionary Access Control
     - 3.1.1 Files
       - 3.1.1.1 Environment
       - 3.1.1.2 Data
       - 3.1.1.3 Expected Outcomes
     - 3.1.2 Directories, Devices, and Named Pipes
       - 3.1.2.1 Environment
       - 3.1.2.2 Data
       - 3.1.2.3 Expected Outcomes
   - 3.2 Data for Condition 2: Security Relevant Aspects of Pathnames
   - 3.3 Data for Condition 3: Process Address Space Isolation

4. Coverage Analysis
   - 4.1 Coverage for Condition 1: Discretionary Access Control
   - 4.2 Coverage for Condition 2: Security Relevant Aspects of Pathnames
   - 4.3 Coverage for Condition 3: Process Address Space Isolation
   - 4.4 Conclusions

5. Test Environment Dependencies

Figure 3. Test Plan Outline for the "access" System Call.

Fact and Rule Set Design. For each system call subject to Security Testing we wrote an associated set of facts and rules using up to 30 possible Prolog predicates that we designed for system call testing. For example, Figure 5 contains a sketch of the fact and rule set for the **stat** system call. In Figure 5, fact "system_call(stat, int, 2, agd)." states that the system call named **stat** returns an integer value, has two arguments, and its security testing type is access graph dependent. We omit the description and rationale of the other Prolog predicates in Figure 5. Figures 6 and 7 show the C program and shell script written by Prolog from the fact and rule set for **stat**. Note that for **stat** the tiny C program just invokes the system call. In contrast, the fact and rule set for **kill** writes two C programs. Figure 8 shows the test results report for **stat**.

Test Driver Design. Figure 9 sketches the test driver design.

5. Experience

Run Time. See Figure 10 for a summary of AIX 2.2.1 Security Testing statistics. On one Model 025 RT with 4MB RAM, from-scratch testing takes 12 hours and 42 minutes. Without recompiling the C test programs, the testing takes 11 hours and 55 minutes, a savings of 7%. On a Model 125 RT with 8MB RAM, from-scratch testing takes 6 hours and 51 minutes, a time savings of 46% over a Model 025. For testing between two Model 125 RTs with TCP/IP, from-scratch testing takes 12 hours and 46 minutes.

Results. Running the security tests produced, in addition to a results report of about 150 pages, about one page of security "lint" relating to the following system calls: **auditlog**, **fcntl**, **open**, **shmat**, **symlink**, and **vmount**. Each test has a (predicted, actual, predicted == actual) outcome triple like sss or ffs, where "s" means success and "f" means failure. The "lint" refers to triples with "f" as the third item. Close review uncovered no "showstoppers," only a few small errors in the test plan.

Extensibility. With the Secure XENIX experience in mind, we designed the security testing system to scale eventually to B2 security testing. We handled in-process changes to the system calls from the AIX development organization, including **vmount** and **uvmount**, in a short time. Also, our original effort included AIX Distributed Services (DS) [Sauer 87], which we set aside when a decision was made not to include it in the TCB of AIX 2.2.1. For DS, remote objects add new security test conditions.

1. Introduction

The synopsis of system call **stat** is

```
#include <sys/stat.h>

int stat(path, buf)
char *path;
struct stat buf;
```

System call **stat** gets the status of the object specified by the last component of the *path*. The second argument to **stat** is a buffer for storing the information about the object. System call **stat** is not affected by the discretionary access permissions on the actual object, but requires search permission on the directories specified in the *path*.

2. Test Condition Selection

The following test conditions are derived for **stat** based on the access graph dependency of **stat** on the **access** system call.

Test Condition 1: Demonstrate that **stat** will fail if any directory component in the path is not searchable.

Test Condition 2: Demonstrate that **stat** will fail if the *path* or *buf* parameter points to a location outside the calling process address space.

3. Test Data Selection

This section describes the test environment, the test data, and the expected test outcomes.

3.1 Test Condition 1: Every Directory Component in the Path Should be Searchable

3.1.1 Environment

To create the test environment, user U1 logs in with a primary group G1. Create the objects in Table 1 below for this test.

Path	Mode
$HOME/dir1	777
$HOME/dir1/file1	644
$HOME/dir1/dir2	666 (N.B., not searchable)
$HOME/dir1/dir2/file2	644

Table 1. Environment for Test Condition 1.

3.1.2 Data

User U1 runs a test program that issues system call **stat** on the objects specified in Table 2.

3.1.3 Expected Outcome

Path	Result
$HOME/dir1/file1	S
$HOME/dir1/dir2/file2	F (because dir2 is not searchable)

Table 2. Expected Outcome for Test Condition 1.

3.2 Test Condition 2: Process Address Space Isolation

Refer to the Process Address Space Isolation test plan.

4. Coverage Analysis

The coverage analysis is based on the access graph dependency and the dependency is proved in Test Condition 1. The test cases with different access modes are limited in number because extensive testing of the AIX access modes has been done in the **access** system call testing. Test Condition 1 constitutes the discretionary access control part of the security policy. Object reuse testing is not relevant for **stat** as there is no direct memory allocation and deallocation. The *path* and *buf* could point to an address location outside the calling process address space and the test is covered in Test Condition 2. The coverage analysis for **stat** justifies the selection of test conditions and test data. The total number of test cases is 2.

5. Testing Environment Dependencies

To create and delete the test environment, system calls **creat**, **mkdir**, **rmdir**, **unlink** must be used in any test program for **stat**.

Figure 4. Security Test Plan for the "stat" System Call.

```
system_call(stat, int, 2, agd).  /* agd = access graph dependent */
argument(stat, path, 'char *', 1).
argument(stat, '&buf', 'struct stat', 2).

testing_to_do(stat, [agd]).

change_groups(no, []).
sys_call_names([user1]).

testing_id_list([uid1gid1]).
get_priv_names([owner]).

/* Omitted Prolog code to write C program goes here. */
/* Omitted Prolog code to write shell script goes here. */

objects_to_test(stat,yes_objs,[dir,file]).
write_to_pipe(no_wr,[]).
objects_in_subdir(yes_subdir,file,[file1,file2]).
objects_in_subdir(yes_subdir,dir,[dir2]).
second_user_owner(no_second,_,[]).
test_range(no_range,_,[]).
get_test_type([test1]).
object_priv(dir, owner, [rwx,rw]).
object_priv(dir, group, [rwx,rw]).
object_priv(dir, other, [rwx,rw]).
object_priv(file, owner, [rw,rw]).
object_priv(file, group, [r,r]).
object_priv(file, other, [r,r]).

expected_outcomes(file, owner, owner, test1, [file1(s), file2(f)]).
```

Figure 5. Sketch of Fact and Rule Set for the "stat" System Call.

```
#include <sys/stat.h>

main(argc, argv)
        int argc;
        char *argv[];
{
        char *path = argv[1];
        int result;
        struct stat buf;

        result = stat(path, &buf);
        printf("%d.\n", result);
        exit(0);
}
```

Figure 6. C Program Written to Test the "stat" System Call.

```
/u/user1/stat.exe  /u/user1/dir1/file1
/u/user1/stat.exe  /u/user1/dir1/dir2/file2
```

Figure 7. Shell Script Written to Test the "stat" System Call.

```
Test Results for the stat System Call

For each of the matrices below, the first s/f for
each individual entry represents the expected
outcome, the second represents the actual outcome,
and the third represents the result.


Test Condition 1 for files:

logged on as U1.[G1]:
                                          Results
        For /dir1/file1:                  sss
        For /dir1/dir2/file2:             ffs

The total number of logins is 1.
The total number of stat system calls is 2.
```

Figure 8. Test Results Written for the "stat" System Call.

```
Overview
(0)     Configure the test driver.
(1)     Do pretesting startup work.
(2)     Test system calls.
(3)     Do posttesting shutdown work.

(0)     /* Configure the test driver. */
        specify with or without TCP/IP;
        specify which system calls on to-do-list;
        specify other configuration data;

(1)     /* Do pretesting startup work. */
        save start time; zero the statistics;
        create directories for programs, shell scripts, and results;
        update users and groups with "adduser" command;
        compile Prolog library procedures and test driver;

(2)     /* Test system calls. */
        for (each system call x in to-do-list)
        {
                compile fact and rule Set for x;
                set subject user name and primary group;
                delete system call test env't;
                create system call test env't objects;
                create C test programs and compile them;
                create per-test-condition shell scripts;
        (2x)    execute tests, collect statistics, save results;
                abolish fact and rule set for x;
        }

(2x)            /* Execute tests for system call x. */
                for (each x-relevant object access mode a)
                for (each x-relevant subject user-group identity s)
                for (each x-relevant test condition c)
                for (each c-relevant expected outcome matrix m)
                for (each m-relevant individual putcome o)
                        test(x, a, s, c, m, o);

(3)     /* Do posttesting shutdown work. */
        save end time;
        /* print statistics; */
        /* print test results report; */
```

Figure 9. Sketch of Test Driver Design.

```
Security Test Plan Statistics
     12   person-months to complete test plan book
   ~500   pages in "AIX 2.2.1 Security Test Plan for System Calls" book
     13   pages in longest test plan ("access")
    124   system calls
    119   security test plans: one plan (not two) for
                         chown[x], msg[x]rcv, open[x], read[x], write[x]
     27   "not security relevant" system calls
      6   "independent" system calls"
     25   "access graph dependent" system calls
     28   "access check dependent" system calls
     39   "privileged" system calls
     57   "process address space isolation" system calls
      8   "object reuse" system calls
 ~2,100   total system call invocations in test plan book
  1,320   maximum system call invocations in one test plan ("access")

Security Test Run Statistics
 12h42m   elapsed time, all system calls, one Model 025 RT, 4MB, from scratch
 11h55m   elapsed time, all system calls, one Model 025 RT, 4MB, no recompiling
  6h51m   elapsed time, all system calls, one Model 125 RT, 8MB, from scratch
 12h46m   elapsed time, all system calls, two Model 125 RTs, from scratch,
   ~150   pages in results report
  2,278   system call invocations
    402   objects created (files, directores, ...)
    240   Prolog-written C programs
    369   Prolog-written shell scripts

Security Testing Fact and Rule Set Statistics
      8   person-months to complete security testing system and per-system call facts
  2,310   facts and rules in Fact and Rule Sets of system calls
    269   rules (166 rules in library, 103 rules in test driver)
  2,278   "logins" simulated by "su"
```

Figure 10. AIX 2.2.1 Security Testing Statistics.

6. Conclusions

Security Testing has been a subtle, expensive, and potentially error-prone process; it is also exceedingly tedious, unglamorous, and thankless. However, it is a requirement of trusted computer systems [TCSEC 85]. It has been subtle because sufficient test conditions are not pervasively understood, little has been published until recently ([Gligor 87b, NCSC 88]), and most testing people falsely assume without coverage analysis that sound Functional Testing covers Security Testing. It has been expensive because it has been labor intensive. It has been potentially error-prone because it is tedious and it has been previously a largely manual process. Our goal, which we achieved, was to decrease the cost significantly and to move toward an error-free process by more automation.

We conclude from this work and the Secure XENIX experience that, compared to writing and maintaining security test programs manually written in C, comprehensive security testing of UNIX system calls with an automated security testing system is overall

- *less expensive*,
- *less error-prone* due to a uniform and automated test plan-program correspondence, and
- *easily extensible* to new system calls, to changes in test conditions, and to B2.

Acknowledgments

For helpful discussions on this work and on the ideas in this paper, we thank Abhai Johri, Scott Chapman, Sohail Malik, Ken Witte, and John Langford. As a stepping stone, we thank the Security Testing team of Secure XENIX. We thank Scott Prevost of CMU and Ruthellen Leventer of Cornell for initial pilot project work in summer 1987. For supporting our efforts, we acknowledge: at IBM SID, Gary Kennedy, Paul Kirby, and Don Del Giorno; and at IBM AWD, Dan Cerutti, Khoa Nguyen, Bob Willcox, and Rose Ann Krauter. Finally, this work would not have been done without the support, encouragement, and intellectual challenge of Sekar Chandersekaran.

References

[AIX CR] *IBM RT PC AIX Operating System Commands Reference* (1988).

[AIX TR] *IBM RT PC AIX Operating System Technical Reference* (1988).

[Cugini 89] Cugini, J. A., "The Inclusion of Set Theoretical Concepts in Prolog," *1989 International Conference on Computers and Communications*, Phoenix, Arizona, pp. 528–32 (March 1989).

[Gligor 87a] Gligor, V.D., *et al.*, "Design and Implementation of Secure XENIX," *IEEE Trans. on Software Engineering*, Vol. SE–13, No. 2, pp. 208–221 (February 1987).

[Gligor 87b] Gligor, V.D., *et al.*, "A New Security Testing Method and Its Application to the Secure XENIX Kernel," *IEEE Trans. on Software Engineering*, Vol. SE–13, No. 2, pp. 169–183 (February 1987).

[Hecht 87] Hecht, M.S., *et al.*, "UNIX without the Superuser," *1987 Summer USENIX Conference*, Phoenix, Arizona (June 1987).

[Hecht 88] Hecht, M.S., *et al.*, "Experience Adding C2 Security Features to UNIX," *1988 Summer USENIX Conference*, San Francisco, California, pp. 133–146 (June 1988).

[Leventer 87] Leventer, R. and Prevost, S.A., "A Knowledge-Based Approach to Software Security Testing," unpublished manuscript, IBM Systems Integration Division, Gaithersburg, Maryland (August 1987).

[NCSC 88] National Computer Security Center, *A Guide to Understanding Security Testing and Test Documentation in Trusted Systems*, 9800 Savage Road, Fort George G. Meade, Maryland 20755–6000, draft (December 1988).

[Pesch 85] Pesch, H., Schnupp, P., Schaller, H., and Spirk, A.P., "Test Case Generation Using Prolog," *IEEE Trans. on Software Engineering*, pp. 252–258 (April 1985).

[Quintus 86]	*Quintus Prolog User's Guide*, Version 8, Release 1.5 for IBM RT PC, and *Quintus Prolog Reference Manual*, Quintus Computer Systems, Inc., Mountain View, California (October 1986).
[Sauer 87]	Sauer, C.H., Johnson, D.W., Loucks, L.K., Shaheen–Gouda, A.A., and Smith, T.A., "RT PC Distributed Services Overview," *Operating Systems Review*, Vol. 21, No. 3, pp. 18–29 (July 1987).
[TCSEC 85]	National Computer Security Center, *Trusted Computer System Evaluation Criteria*, DoD 5200.28–STD (December 1985). This reference is commonly called the *Orange Book* after its cover color.
[Tsai 89]	Tsai, C.R., *et al.*, "A Trusted Network Architecture for AIX Systems," *1989 Winter USENIX Conference*, San Diego, California, pp. 457–471 (February 1989).

FACTORS AFFECTING APPLICATION PORTABILITY TO A B1 LEVEL TRUSTED UNIX

Jon F. Spencer
Senior Program Engineer
Addamax Corporation

Jackie McAlexander
Manager - Technical Support
Addamax Corporation

ABSTRACT

With the increasing use of operating systems which have been modified to meet multi-level security requirements comes an increasing need to port standard UNIX (r) applications to these platforms. This papers discusses major changes made to target the B1 level of trustedness. It shows that vendor implementations may vary, and that these variations affect the portability and performance of existing applications.

The paper takes the UNIX mail program and discusses several implementation methods on a B1 system.

INTRODUCTION

Last year, pressure to tighten security on almost all government data processing operations converged with pressure to standardize on a single operating system, and the result was a dramatic increase in the specification and purchase of "trusted" UNIX systems. Federal contracts, worth hundreds of millions of dollars, were awarded for systems running versions of UNIX modified to meet at least a class C2 level of trust (as defined by the Department of Defence "Trusted Computer System Evaluation Criteria" [TCSEC]). Under the terms of these contracts, most of the acquired systems must quickly evolve to class B1 or B2 multilevel security.[1]

It should be pointed out, that, at the time of this writing, only one UNIX based operating system (Gould's C2 version of UTX) has been formally evaluated by the National Computer Security Center (NCSC, or Center). One of the tasks of the NCSC is to perform evaluations of proposed trusted systems (including both the hardware and the software taken together to comprise an evaluable *system*), and it is the successful completion of this formal evaluation by the Center that conveys true *trusted* status on a system.

[1] To review, the TCSEC classes are levels of features and assurances which are ordered from least trusted (D level) to most trusted (A1 level). The "C" levels require discretionary protection of data, while the B levels add mandatory protection. At the higher B levels (B2 and B3), the emphasis shifts from features to assurances, or the design and testing of the system.

Several UNIX vendors are either in the design stage or are already offering systems targeted at the B levels. Of these, a few have submitted their products for NCSC evaluation. This paper considers these "designed to be trusted" systems without commenting on their prospects for formal evaluation. Of interest to note is that for reasons of market demand or of compatibility with future products, several vendors have included B2 level features in their B1 products (such as a partial implementation of the B2 concept of *least privilege*). The Center will evaluate at B1 a system which has met *at least* all of the B1 requirements, but has not met *all* of the B2 requirements. This fact must be kept in mind by applications programmers when porting an application to run on B1 systems provided by several different vendors.

Although it is discussed in more detail later in this paper, it may prove helpful here to note that a trusted system is composed of both trusted and untrusted software. A particular trusted component may reside in the kernel, or may be implemented as a command (application). There are no explicit requirements imposed by the TCSEC in this regard, although for performance reasons certain trusted capabilities are much more efficiently implemented in the kernel.

The demand for applications to run on the increasing installed base of trusted UNIX systems will probably be satisfied in three distinct ways. First, to make multilevel use of the security features implemented in these multilevel systems, new trusted applications software may need to be written (e.g., a multilevel mail service).

Probably more often, existing applications will be analyzed and modularized to segregate the security related components of the code. The portions of the code which do not interact with the security enforcement components of the trusted system will be reused essentially as-is, while the components of the application which must be trusted will be reimplemented.

But many times, standard UNIX packages, such as word processing, simulation and graphics software, will just be ported as untrusted applications, with possible minor modifications to the environmental setup procedures. This will be a logical and important use of trusted UNIX. In fact, it is precisely because such care is taken with the features and assurances of a trusted operating system, that untrusted software can be hosted without subverting security controls. This property opens trusted systems to the world of UNIX off-the-shelf solutions. We expect that, as the number of trusted UNIX installations increase, so will the demand to port many (if not most) standard UNIX applications to these platforms.

What should application programmers, who may not be familiar with trusted system design, consider when porting these applications? What features of standard UNIX will they find changed?

Unfortunately, the answer is: it depends on the vendor's implementation of the trusted UNIX operating system. While several groups, including POSIX, OSF and X-OPEN, are working towards security standards, a complete, standardized trusted UNIX (or POSIX) will not emerge for several years. In the meantime, most vendors offering a "secure" product make their own, proprietary changes to the UNIX kernel and program interfaces. Implementations vary depending on the security level being targeted and the individual vendor's design.

This paper looks at changes commonly made to UNIX when targeting a B1 level certification. It discusses popular approaches and proposed standards and looks at which approaches might facilitate or complicate the porting of existing untrusted applications. An example is presented which helps to demonstrate difficulties which may be encountered,along with possible solutions.

BACKGROUND

Since this paper will examine factors which affect the porting of untrusted software into a trusted UNIX environment, a brief review of what distinguishes trusted from untrusted applications is presented.

One of the basic concepts of trusted systems is the Trusted Computing Base (TCB), a term which has no standard UNIX equivalent. The TCSEC defines the TCB as the "totality of protection mechanisms within a computer system" and the "components that together enforce a unified security policy...".[2] In UNIX, the major pieces of the TCB are the operating system kernel, device drivers, and STREAMS or Socket modules (if present), and a set of trusted commands. However, since any process that can subvert the basic security policy of the system must be trusted, other commands (applications) will be included.

Basically, any application which will handle data at more than one security level, use privileges, or alter data that is used to make security decisions, must be ported as trusted software. This is not to say that the entire application should be trusted. Trusted software design principles encourage the use of the smallest possible security relevant *modules* which will become part of the TCB. But a thorough analysis must be done of the entire application to identify each routine or subroutine and determine the subset which perform a security-related function. These modules will have to be isolated and rewritten as trusted code, thereby becoming a part of the TCB.[3]

There are many examples. Utility packages that perform functions like backup will have to be trusted to maintain security labels and ACLs (access control lists), and to be able to access data at multiple levels. Database systems will have to be trusted if they handle data at several levels concurrently. Even a seemingly generic application like mail will have to be at least modified if mail is to be sent to the same user at more than one level, or will have to be made trusted if copies of a single mail message are to be delivered to users at multiple levels (see the example application port).

Creating new trusted applications, or modifying existing software to perform trusted functions, requires a thorough understanding of security principles and of the security policy of the underlying operating system. Specification, design, configuration control, testing and documentation must follow special, sometimes rigid, guidelines.

However, if an application will handle data at only one level (or perhaps one level at a time) and if it can be configured to run without privileges, porting it as untrusted should suffice. Even if the system itself handles users at many levels, these untrusted applications can trust the controls of the TCB to secure their data (or, more accurately, the TCB can assure that the untrusted application does not, will not and cannot violate the system security policy).

CHANGES TO UNIX AT B1

The B1 level was chosen for discussion because it is the level where the most significant functional changes occur to a trusted UNIX operating system. Most standard UNIX systems can be adapted to C2 requirements with minor enhancements to existing features and the addition of audit trail capabilities. The B2 level adds mostly "assurances" to features that are added in lesser levels. Also, changes at B2 usually occur too deeply in the operating system to greatly affect applications.[4]

It is at the B1 level that data begins to be tracked according to sensitivity labels, mandatory access controls are enforced based on these labels, and discretionary access to data is usually tightened. At B1, system calls can be substantially modified and many new calls may be added. A program interface may be established at B1 which, if well designed, can remain constant through higher levels. Programs written for B1 can therefore migrate to systems of higher trust with little or no modification, assuming that the security models for the source and destination systems are compatible.

It is beyond the scope of this presentation to discuss any one trusted UNIX feature in detail, since each is a topic in itself. Instead, we will concentrate on general changes in UNIX functionality which may affect standard UNIX applications.

In order of most potential impact on applications, they are:

1) Addition of mandatory access controls
2) Enhancement of discretionary access controls
3) Changes to Superuser and setuid
4) Treatment of devices and special files

For further information on any topic, please see the attached bibliography.

ADDITION OF MANDATORY ACCESS CONTROLS

Mandatory access controls (MAC) are entirely new to UNIX and, although the TCSEC explicitly defines the content and use of mandatory security labels, they are implemented differently from vendor to vendor.

• *MAC Labels*

By definition, all B level systems must restrict access to data based on sensitivity labels. The labels are applied to both subjects (UNIX processes) and objects (files, directories and other "data containers").

The label consists of two parts:

- The hierarchical classification describing the sensitivity of the data
- A non-hierarchical set of compartments describing the use of the data

Since these labels are *mandatory*, they are automatically assigned to processes (usually based on the login label of the user) and on objects which are created as a result of those processes.

Depending on the vendor's implementation, MAC labels (and enhanced discretionary access information) may be stored in an external database or, for performance efficiency, directly in the inode of the file. Since calls to check MAC are made each time access to an object is attempted, efficient implementation of MAC is considered one of the two most significant performance factors in a B level trusted system (the other being auditing).

• *MAC Rules*

The ability to access data or execute programs is controlled by the interaction of the subject (process) label and the object label. Again, the TCSEC is explicit in its definition of the rules of access which are based label *dominance* (see *MAC Relationships* below).

The simple intent of all TCSEC mandatory controls is to restrict access of data to users with sufficient clearance to see it. So an obvious restriction of MAC is that a user's (subject) label dominate a file's (object's) label for read or execute access to succeed. In this manner, the flow of information is always *from* a source of lower clearance *to* a sink of higher clearance.

Data being written is also strictly controlled by MAC to prevent the *downgrading* of data. Security policy may allow data at a lower label to be written "up", to a higher label, but never written "down" (again enforcing the *from ... to ...* rule above). This basic MAC restriction is applied to all levels of data and will affect both labels being changed (i.e. always must be changed to a higher label) and data being copied (i.e. must always be copied or moved to a higher label).

Among the various implementations, and at various levels of the system (system call, utilities etc.) these basic restrictions are applied in specific rules, but always with the same basic intent - without special privileges, data can at most be upgraded.

In practice, there are many constraints to data being "written up" or a MAC label being changed "up". For instance, if a process were to write to a file at a higher level, that process could not read the file it had just written. And, if an open connection to a file existed when a change to the MAC label occurred, MAC would be violated.

Generally, several additional restrictions to normal operations are usually imposed so that data is not accidentally or intentionally downgraded. For example, users are usually restricted to writing to files whose MAC label exactly matches their process label. And, since directories are also files, processes are usually restricted, at least when creating files, to directories which match their session labels, and when writing to existing files, to directories which their session labels dominate.[5]

• *MAC Relationships*

A MAC label has two distinct components. The first is its *class*, which forms a well ordered set, where the relationship between two classes is always less than, equal to, or greater than. The second component is a *set* of compartments or *categories*. This set must be a (improper) subset of the full set of categories allowed for a particular system at a particular time. A label A is said to dominate a label B if:

1. A(class) >= B(class)
2. B(categories) $\supseteq$ A(categories)

From set theory, it is apparent that the relationship between the categories component of two arbitrary MAC labels is not well ordered. This aspect of MAC labels is often neglected in discussions of accessing data. Writing or reading "up" or "down" have no meaning for a pair of MAC labels for which no dominance relationship can be specified.

As an example, if A and B are MAC labels, and A does not dominate B, then it does not necessarily follow that B dominates A (i.e., B is not necessarily "higher" than A). For instance, B may be defined as class TOP SECRET with categories ADMINISTRATION, LASERS and SDI, while A may be defined as CONFIDENTIAL with categories PERSONNEL and PAYROLL. Even though B is at a higher class than A, and A is not allowed to access data pertaining to ADMINISTRATION, LASERS and/or SDI, neither is B allowed to access data pertaining to PERSONNEL and/or PAYROLL, although it may only have a class of CONFIDENTIAL.

• *MAC Results*

The obvious result of adding mandatory access controls is denial of access to users who otherwise would have had sufficient UNIX permissions to perform an action. Fortunately, many of these problems can be prevented by proper configuration. In fact, MAC frequently causes more problems that are due to configuration and use of software than to actual code incompatibility.

As a result of MAC restrictions, the type of access to data that an application requires will determine the degree of "trustedness" which must be implemented. The types of access may be generally classified in one of four ways:

1. All data files have the same label as the process.
2. All data files being written are at the same label as the process, and the labels of all data files being read are dominated by the process label.
3. All data files being written are at the same label as the process, but the labels of some of the data files being read are not dominated by the process label.
4. The labels of some or all of the data files being read are not dominated by the process label, and/or the labels of some or all of the data files being written are not equal to the process label.

From the standpoint of the TCB, there are actually only two cases:

A. The security policy regarding MAC is obeyed.
B. The security policy regarding MAC is overridden.

• *MAC Rules Are Obeyed*

Classification 1 falls under this case. An example of this kind of application is an editor such as vi (but see Shared Public Directories). Classification 2 is also contained in this case, and an example of this would be a word processing application. Here, the process and the file being edited are at the same label, but the labels of common files requiring read access, such as dictionaries and source text files, are dominated by the process label.

No portion of any applications in this case need be resident in the TCB. The only security issues relate to the configuration of the system. The system administrator must install the application executables and common files such the the labels of these items are each dominated by the labels of all processes which are to have access to them. Applications in this case should port essentially unmodified.

• *MAC Rules Are Overridden*

Classifications 3 and 4 are clearly in this case. Whenever any attempt is made to internally override the system security policy, all such attempts must occur in, and must be mediated by, the TCB, and this trusted code must be carefully reviewed to ensure that no violation of the security policy can manifest itself outside of the TCB.

What does this really mean? It means, for example, that the trusted portion of an application may temporarily raise the label of a process to read a protected password file to determine at what labels a user may receive mail, or may temporarily lower the process label to write to a FIFO to communicate with a daemon (or, in some implementations, request that MAC be ignored during those operations). In any event, applications for this case will require very careful examination to ensure that the normal rules of label dominance are enforced for the untrusted user.

• *Implementation Considerations:*

Before porting or configuring an application on a trusted UNIX with Mandatory Access Controls, the security policy of the system must be completely understood. For B1 systems, the TCSEC requires that this be spelled out in the "security policy model" which should be available, along with other required documentation, from the vendor.

The portability of applications will be affected if new system calls are added to check MAC. If the trusted UNIX being targeted checks mandatory access controls with extensions to standard UNIX system calls (such as within "s5access"), error codes being returned must be checked to ensure consistency with the appropriate standard UNIX baseline.

• **Shared Public Directories and Setuid**

Even applications which are run in a strictly single-level mode can present problems on a multi-level system. Two commonly used practices, writing to shared public directories and using setuid (especially to root) to gain privileges, are modified in some fashion in all trusted systems. A well designed implementation will provide backwardly compatible substitutions.

In a multilevel system, MAC restrictions prohibit the use of shared public directories, such as /tmp. Without a backwardly compatible substitute, even simple applications like "vi" break. The most common approach solution to this incompatibility is the use of 'partitioned directories'.

A partitioned version of the directory /tmp, for example, will really consist of many transparent subdirectories /tmp/[MAClabel]. Each process uses the /tmp/[MAClabel] of its current process label. Processes never see the label portion of the path. They are also protected from seeing files stored in the parent directory (e.g., /tmp) at any other label.

Most trusted UNIX vendors have implemented partitioned directories, and some form of partitioned directories will almost certainly be included in trusted UNIX standards.[6] [For those readers horrified at the thought of the simultaneous creation of 2^n transparent subdirectories whenever a partitioned directory were created, the normal implementation of this approach does not actually create the subdirectory for a particular label until the parent directory is traversed by a process at that label.]

• *Set UID/GID*

The set of privileges commonly referred to as setuid are not, strictly, a MAC issue. However, their use in B1-level trusted systems will be discussed in this section.

UNIX folklore often cites setuid as the root of all security evils. Certainly many systems programmers wrote privileged programs not designed to uphold basic security principles. Also, the mechanism whereby users set "bits" on an executable file so that anyone executing that file is given its effective user and/or group identity can allow the "propagation of discretionary access controls" - expressly discouraged in the TCSEC.

However, absolute removal of setuid leaves UNIX without a powerful and frequently used feature. An early version of trusted UNIX, which completely eliminated setuid, quickly added it back as the vendor realized that most existing applications would not work.[7] Without some logical equivalent to the setuid to root concept, or to the B2-level concept of *least privilege* (discussed later), there would be no way to convey the attribute of being a part of the TCB to an application.

• *Implementation Considerations*

For ease of porting, the target trusted system should allow flexibility when configuring setuid. Several implementations allow choices on a per-file system basis. Most distinguish between setting and observing setuid bits and between root and non-root privileges. At least one implementation immediately removes setuid bits if the owner, group or if the data in the file is altered (virus protection).

If the application to be ported uses setuid, the vendor implementation should be studied carefully for compatibility.

• Compatibility With Standard UNIX Objects

Files and directories being transferred (imported) to the trusted system from standard UNIX will not have mandatory (or extended discretionary) labels. The trusted system should provide a way to handle these non-labeled objects, both individually and in bulk.

• *Implementation Considerations*

In most systems, MAC labels can be applied to files when they are imported into the trusted system.

In addition, some implementations provide a standard UNIX compatible file system mount capability, where the mandatory access label of the mount point is applied to all objects in that file system. Using such a system, it is possible to use existing disks and tapes on the trusted system without conversion or modification.

• Compatibility With New File System Types and Attributes

Along with providing compatibility with existing UNIX file systems, it may be highly desirable that the target trusted UNIX allow for extensions and changes as the definition and use of security labels evolve. This would allow applications to handle data with different label types on the same system, without modification of the applications.

New procurements are already specifying formats that extend the basic MAC labels described in the TCSEC. Among other things, these new labels must describe special handling restrictions and other caveats. Standards, too, while they may not ever specify the internal representation of labels, will certainly cover label formats on data being exchanged among secure systems.

• *Implementation Considerations:*

If the implementation supports different label formats, the vendor usually makes use of a "file system switch", much like the character/block device switch. This system makes use of the fact both System V and BSD separate their file systems into a single, upper layer, which lies closest to the system call interface, and a lower layer, which lies closest to the hardware. Each mounted file system has a single lower layer, called its "file system type". A kernel table indicates which lower-level processes are to be used for each file system type.[8]

Using this method makes label formats transparent to application programs.

• MAC Summary

The addition of mandatory access controls is probably the greatest change to operating system functionality when migrating to B levels of trusted UNIX. Depending upon the degree to which an application requires read and/or write access across label boundaries, MAC effects on ported applications can vary from almost invisible to almost "impossible".

But, by understanding which applications can be maintained as essentially untrusted code and by understanding the configuration restrictions of such code, many off-the-shelf UNIX packages can be used, in limited ways, without modification.

EXTENSION OF DISCRETIONARY ACCESS CONTROLS

Discretionary access controls (DAC) are those that are applied at the 'discretion' of the individual, non-privileged user. In standard UNIX they are the /user/group/other permission bits of files, directories and System V interprocess communication (IPC) objects.

Requirements for DAC begin at the C2 level. For classes C2 through B2, the TCSEC specifies that they "be capable of including or excluding access to the granularity of a single user". It further offers the examples "e.g. /self/group/public controls, access control lists" (TCSEC, Section 3.1.1.1).

Whether UNIX permission bits alone satisfy this TCSEC requirement has long been debated, since the /user/group/other protect bits easily comply with the first example but do not generally allow exclusion to the granularity of a single user. It is now almost universally acknowledged that, at least at the B levels, simple protect bits will not suffice. Therefore, standard UNIX is usually augmented with some type of access control list mechanism, beginning at B1.

• *Implementation Considerations:*

Implementations of DAC vary widely among vendors. A few vendors have extended the group mechanism in non-standard ways to provide 'pseudo' control lists as a temporary bridge to full access control lists (ACLs). One vendor offers modified privileges (discussed below) instead of access control.

But most vendors who intend to track standards and qualify for higher levels of trust have modified their B1 products to provide full access control lists. Certainly: 1) ACLs will definitely be needed at higher levels 2) the NCSC strongly encourages this approach and 3) POSIX has several good proposals for access lists.[9]

Because UNIX has existing discretionary controls, it is important that the mechanism in the new, trusted system be as backwardly compatible with standard UNIX as possible. The goal should be to preserve both code and function portability. This can be accomplished in several ways.

First, access control lists are most easily understood by UNIX users if their format is similar to traditional protections. An ACL whose entries consist of users and groups followed by the familiar UNIX read/write/execute format maintains excellent compatibility.

Next, a well designed access control system should not remove, disable or change the meaning of standard UNIX protect bits. Additional information should be used to augment, not override, standard UNIX protections.

Existing applications sometimes use protection bits for locking files, as an example. The mode of a file may be changed several times, usually from less to more restricted, as it opened for read and then for write by a process. At the conclusion of the operation, the original mode will need to be restored. The existence of an ACL should not break these operations. Similarly, these operations should not destroy any ACLs that have been established for that file.

A commonly implemented system which uses UNIX protect bits to mask an ACL has these properties. Every file on the system has traditional protect bits and an ACL, although the ACL can be empty. If the ACL is empty, the UNIX protect bits work as they do in standard UNIX. If the ACL is not empty, the group protect bits are replaced with a search of the ACL for specific inclusions and exclusions but, in no case, can greater permission be given than is given by the standard UNIX group protect bits. As the group protect bits are changed, to lock the file, for instance, permission may be denied to other users, even if they appear on the access control list.

Finally, to the greatest extent possible, access controls lists should not degrade system performance. This is more likely to be achieved if the access control information, like the UNIX protect bits, is kept in the inode of each file.

• ACL Summary

In summary, existing applications should transfer easily to a well designed system which makes use of Access Control Lists. Such a system should satisfy the discretionary access requirement through the B3 level.

PRIVILEGE

It is of interest to note that for a system to be certified "B1" by the NCSC does not precisely state the set of properties and capabilities that the product has, but rather the *minimum* level of trust that the product does provide; namely, that level of trust specified as B1 by the TCSEC. Indeed, for reasons of compatibility with future products (as well as for other reasons), some vendors have included capabilities of higher levels in their B1 system. One such capability is privileges.

In trusted operating systems, some programs require the ability to perform operations that, if done improperly, could compromise the security of the system. In standard UNIX there is only one such privilege, namely "superuser" (root). The privilege can be acquired either by initiating a root session (login, su), or by executing a setuid root application.

At the C and B levels, the TCSEC requires that the privileged processes be part of the TCB (that is, the part of the system that is evaluated) but at B1 no restrictions are placed on the scope of those privileges.

At the B2 level, the concept of "least privilege" is introduced, being the splitting of the superuser privilege into subsets of privileges, each of lesser capability, whose union is essentially that of superuser. These privileges are normally an attribute of the process and are independent of the owner of the file. At higher levels, certainly, "root" as we know it in UNIX will go away.[10]

Replacing superuser with a process privilege scheme affects the portability of many UNIX applications and system utilities, especially those which internally check for superuser by testing that the user ID is 0.

• *Implementation Considerations:*

Trusted UNIX standards for least privilege (at B2 and above) are evolving, but current vendor implementations vary widely. While some vendors have introduced a sometimes incompatible system of privilege even at the B1 level, most implementations attempt to preserve the traditional concept of "root", at least at B1, and at least until standards are more fully evolved.[11] Since compatibility can be so greatly affected, this seems to be a sensible approach. A compromise, whereby separate privilege is given to the new security features of a trusted system, has been considered and should not affect software portability.

Security can be increased, however, even when giving full privileges to root. Some systems subject privileged process to the same MAC (and DAC) restrictions as non-privileged processes when executing standard UNIX system calls. Of course, privileged processes may be able to change their security levels and may be able to change the labels of files, but this is generally done with new, trusted system calls. If only those areas of a program which execute the new system calls could override the basic access control of the system, security relevant code could be made more modular. Then, only these modules would required the most careful design.[12]

To maintain compatibility with higher levels of trusted UNIX, as they evolve, applications should avoid trying to determine whether they are invoked as a privileged process (e.g., examine their effective user ID). Instead, they should simply assume they are invoked as privileged where appropriate. If they are not, then the system calls requiring privilege will fail, and the program can then take appropriate action.

It is seldom an effective security measure for a program itself to refuse to perform an action unless it is privileged. If is is privileged, the test is vacuous. If it is not privileged, then it can do no more harm than any other unprivileged process.[13]

If trusted programs must check their own privilege, it should be done in a highly modular fashion because these checks will likely have to change as standards evolve.

• Privilege Summary

There is no clear consensus among vendors on when to begin to implement B2 least privilege. The overlap of mechanisms for privilege with the mechanisms for access control has been confused, at least in the marketplace. It seems fair to say that both privilege and access control lists will be adopted by most vendors, especially at higher levels.

Of the two, however, until standard directions are clearer, least privilege has potentially the greatest impact on portability.

DEVICES AND SPECIAL FILES

A "device" in a UNIX system is an abstraction and is used to cover all objects accessed by referencing device special files. In some cases, these data objects are physical devices or portions of physical devices such as modems, ports, or disk partitions. In the case of pseudodevices, they can be quite different (including /dev/null where there is no data storage at all).

Devices may be characterized as multi-level (i.e trusted labels can be placed on the device) or single-level (i.e no such label exists). Multi-level devices are generally storage devices such as disks and tapes, while single level devices include printers, terminals and modem ports.

To comply with the TCSEC, all physical devices which are used as data containers must be subject to mandatory and discretionary access controls.

Unfortunately, more than one device special file can be mapped to the same physical device. If access were enforced by simply applying MAC and DAC to the device special file, more than one set of access controls could be present on each object. This is prohibited in a trusted system. The TCB must assure that, at anytime, only one set of access controls apply to each device and that the operations that maintain these controls are clear and free from obscure side effects.[14]

• *Implementation Considerations:*

Although trusted UNIX standards have not evolved in this area, they must eventually address a mechanism to ensure a single set access information for each physical device. In the meantime, vendors who need to be evaluated by the NCSC have adopted such mechanisms individually.

Fortunately, few untrusted applications rely on multiple access paths to the same device. However, those porting applications which use "chmod" for control access to a device should be aware that this mechanism may no longer work and devices may be inaccessible that appear to be accessible via MAC and DAC

Instead of using the device special file to control access, it is desirable to have a system of physical device allocation to maintain "controlled" use of units such as tape drives and modem lines. Most vendors add a scheme for device allocation.

Fortunately, device allocation is usually done at time of use, via command line rather than program interface, and so the differing user interfaces are not often a porting consideration.

Pseudodevices, although identical in their use of device special files, are used for many purposes other than I/O. They are so varied that there is usually a special security policy for each. For instance, /dev/null can be left unrestricted since it actually contains no data. Reads from /dev/kmem, on the other hand, must be strictly controlled (reserved for privileged processes).

If the application to be ported uses pseudodevices, the individual implementation will have to be studied. In general, applications that use pseudodevices, like other untrusted applications, will be easiest to port if they are configured to operate at a single level.

An Example Application Port: Mail

As an example of the application of issues that have been presented in this paper, the porting of a relatively simple command into four different security environments is briefly presented. The UNIX command *mail* has been chosen, since its use is rather well known throughout the UNIX community.

The four security environments are:

1. Mail may only be exchanged among users at a single label system-wide.
2. Mail may be delivered to the same user at more than one label.
3. Mail may be delivered to the same user at more than one label, but it is always delivered at a label the user may login at.
4. Mail may be delivered from any label to any label, with dynamic limitations configurable by the system administrator.

• *Assumptions*

It is assumed that the system to be ported to is a straight B1 system with no higher level capabilities available. It is further assumed that partitioned directories are available if desired for use.

Example 1. Mail may only be exchanged among users at a single label system-wide

If mail is restricted so that only users at one single label may exchange mail, then there is no need to make any modifications to the mail command. The single label is the label of the /usr/mail directory. The reason for this is that mail is currently protected by having mail own /usr/mail with group mail, having the mode of /usr/mail be 775, having mail itself be a setgid to mail, and creating the mailbox for a user in the /usr/mail directory with the owner being the user, the group being mail, and the mode being 770.

From this scheme, mailboxes may only be created at the label of the directory, and thus mail can only be added to the mailboxes at that same label. The implication is that all users who were allowed to use the mail system would have to be able to login at that label, and could only use the mail system during sessions with that label. It also implies that the label on all users home directories who use mail must be equal to the label on /usr/mail, so that they may save their mail.

While this is not necessarily an optimal situation, it is workable, and it is an example of how an untrusted application can be successfully ported to a trusted system with no modifications.

2. Mail may be delivered to the same user at more than one label.

Since mail is not required to cross label boundaries during its operation, this facility can be implemented without making mail a part of the TCB. However, the restrictions which prohibited the application in Example 1 from delivering mail at multiple labels, and prevented the user from reading mail at multiple labels, must be overcome.

The first restriction is that mail can only be delivered at the label of the /usr/mail directory. This can be overcome by making /usr/mail a partitioned directory. In this way, mail may function with the same mode bits on both the mail executable and on the mailboxes. Each user will have a mailbox created at each distinct label at which mail is delivered for that user.

The second restriction, that the user can only read mail at a single label, that of the user's home directory, can be solved in a similar manner. The users mbox, saved mail file and dead letter file can be renamed and moved to the /usr/mail partitioned directory. The new name for the files will be similar to the old name, with the user's username prepended to it.

This extension of the mail capabilities requires only minor changes in the mail program itself. Mail need only be modified to construct the file path names differently when sending, reading, and checking for mail. A disadvantage of this approach is that it is possible for a user to receive mail at a label which the user cannot login at. This would result in that mail being "lost", in that the user could not receive it. Additional utilities could be written to occasionally search for such mail files, and to handle them in some appropriate manner, thus avoiding the buildup of this "mail entropy".

3. Mail may be delivered to the same user at more than one level, but it is always delivered at a label the user may login at.

This example is only slightly different than example 2, but it is the first one at which the mail command has a TCB resident component. In order to ensure that the mail being sent can be received by the intended recipient, the mail program must compare the mail label to a list of valid login labels for the user. This information is normally kept in a protected password file. Whatever the label of this file, unless it is at a label which is always guaranteed to be dominated by any process label at which the mail command is run, the mail command will need to alter its process label to dominate that of the protected password file so that it can read that file. And even if the protected password file can always be read, the mail command must alter its process label to that of the delivered mail, in order to write that mail into the recipient's mailbox. (Remember, the partitioned directory subdirectory that is seen by the process is the one that matches the process's label.) Thus, the mail command must have the privilege to modify its own process label, which under the assumptions for the target system, means that mail must be setuid root.

In order to prevent the downgrading of data, the label at which mail is delivered must dominate the label of the mail being sent. Thus there are two cases in which mail cannot be sent. First, if the label of the mail strictly dominates all of the recipient's login labels (i.e., the mail label not only dominates all the login labels, but is also not equal to any of them), then to deliver the mail would require a downgrading of the data, which is prohibited by the TCSEC. The second case is where no dominance relation can be formed between the mail label and any of the user's login labels (due to the inclusion in the mail label of a category which is not included in any of the user's login labels which would otherwise dominate the mail label). Again, the mail cannot be delivered.

An interesting restriction to this scheme is that while it is guaranteed that no mail will become "mail entropy", it is not always possible to reply to a mail message, if that mail message was upgraded when delivered. This would be the case if the sender did not have a valid login label which dominated the upgraded label at which the original mail was delivered.

4. Mail may be delivered from any label to any label, with dynamic limitations configurable by the system administrator.

The last example involves the addition of a mechanism that would allow mail to be delivered to user's at a user's login label which does not dominate the mail's label. Whenever the delivery of mail would require the MAC dominance rules to be violated, rather than refusing to deliver the mail, the mail could be routed to a security administrator who could then manually either approve or deny to reassignment of MAC labels to the mail. This rerouting could be mediated by the contents of a file (say. /usr/mail/MAC.xfer), which would list all the MAC label exchanges which would be automatically allowed, and thus not routed to the security administrator. The mail command should make these reroutings an auditable event.

As can be seen from these examples, if proper attention is paid to the modularization of the application, it should be possible to port many applications in a reasonable amount of time. The difficulty in porting is directly related to the degree to which information must cross MAC label boundaries. It is interesting to note that as the capabilities of the solution increase, so do the areas which must examined and the number and/or size of the modules which are included in the TCB. In addition to the straightforward issue of MAC, the code must also be examined for potential convert channels and outright security holes. An example of such would be any command which was setuid to root, and which invoked another command by relative pathname. In this case, the user could modify the PATH variable to insert his home directory in the search path, and create a shell script with the same name as the command which the setuid to root command was invoking. This shell script could then copy /bin/sh into its home directory, change its permissions and owner to make it setuid to root, and then execute the original command that was requested. In this way, the user would then be able to violate the system, and the fact that this happened would most likely not be discovered. (Such a hole exists in the standard UNIX lp command.)

Conclusion

For users and applications programmers who may need to port software to a trusted UNIX, this paper has discussed major changes which are made to system functionality at the B1 level. It has shown that vendor implementations vary, and that these variations may affect the portability and performance of existing UNIX applications.

In has argued for those systems which maintain backward compatibility with existing code while positioning themselves for higher levels of trust and future standards.

References

[1] National Computer Security Center. Trusted Computer Evaluation Criteria. Department of Defense, DOD 5200.28STD, December 1985.

[2] National Computer Security Center. Trusted Computer Evaluation Criteria. Department of Defense, DOD 5200.28STD, December 1985.

[3] D. Gambel, et. al. Retrofitting and Developing Applications for a Trusted Computing Base. In Proceedings from the 11th National Computer Security Conference. Retrofitting and Developing Applications for a Trusted Computing Base. October 1988

[4] W. O. Siebert, et. al. UNIX and B2: Are They Compatible? In Proceedings from UNIFORUM. January, 1987.

[5] Addamax Corporation. IEEE P1003.6 MAC Proposal. Draft 1. January 1988.

[6] Steve Sutton. The /usr/group and Trusted UNIX. December 1988.

[7] Steven Bunch. UNIX and B2: Are They Compatible? Proceedings from the 10th National Computer Security Conference. September 1987.

[8] Addamax Corporation. B1st Technical Approach. Doc 187-108/2.0. December 1988

[9] Steve Sutton. The /usr/group and Trusted UNIX. December 1988.

[10] Steven Bunch. Least Privilege Mechanism for UNIX. Proceedings from the 10th National Computer Security Conference. September 1987.

[11] K. Addison et. al. Computer Security at SUN Microsystems. Proceedings from the 11th National Computer Security Conference. October 1988.

[12] Addamax Corporation. B1st Technical Approach. Doc 187-108/2.0. December 1988

[13] Addamax Corporation. Security Features Programmers Guide. Doc 288-343/1.0. December 1988.

ULTRIX† Threads

Daniel S. Conde
conde@decwrl.dec.com

Felix S. Hsu
fhsu@decwrl.dec.com

Workstation Systems Engineering
Digital Equipment Corporation
Palo Alto, California

Ursula Sinkewicz
sinkewic@decvax.dec.com

Open Software Components and Resources Group
Digital Equipment Corporation
Nashua, New Hampshire

ABSTRACT

Two groups at DIGITAL recently collaborated on a prototype implementation of threads in ULTRIX, DIGITAL's UNIX®‡ product. Threads, or lightweight processes, provide a means for applications to control complexity and more efficiently utilize uniprocessors and multiprocessors. Uniprocessor benefits can be realized by using threads to overlap I/O operations with computation, and to cope with complex asynchronous run-time environments in a synchronous way. Multiprocessors allow the additional benefit of executing an application's threads concurrently.

A wide variety of designs were considered for ULTRIX Threads, beginning with user-level threads, and evolving to kernel threads. The tradeoffs between user-level threads and kernel threads are discussed and contrasted with other multi-thread systems, both UNIX-based and non-UNIX based. Finally, the kernel, library, and dbx changes are described in some detail, highlighting the tradeoffs and decisions made in our final design.

1. Introduction.

Threads, or lightweight processes are a recent trend in the UNIX community. However, the concept of multiple program counters, or threads of control, in one address space has been in existence for some time. A sampling of these systems illustrates the range of interpretations possible.

- *Threads in the kernel.* When the concept of threads is applied to the kernel, the result is a multi-threaded kernel where the threads run only in kernel mode, and perform kernel functions such as disk read-ahead or write-back[1]. Examples of systems employing threads in

† ULTRIX, ULTRIX-11, and ULTRIX-32 are trademarks of DIGITAL.

‡ UNIX is a registered trademark of AT&T.

[1] "The kernel" here refers to all system components from the UNIX interface down to the lowest-level,

the kernel include DEC System Research Center's Taos, Xerox's Pilot Operating System [2], Stanford's V System [3], and CMU's Mach [1].

- *Kernel-supported user threads.* The purpose of these threads is to support user threads in the same way that processes support user programs. The distinction between this class of kernel threads[2] and those in the previous class is that these threads are created in response to user-mode requests. The kernel may or may not be multi-threaded, but it supports a multi-thread interface. Examples of these kinds of systems include Stellar's Stellix [4], Silicon Graphics Inc's version of System V [5], and Dynix [6].
- *User-mode threads on virtual processors.* In this scheme, processes or kernel threads act as virtual processors and execute any runnable user thread. Inter-thread communication and synchronization are done via shared memory or message passing between the virtual processors. No special kernel support for multi-threading is required. Xerox PARC's PCR [7] and Brown's Threads [8] are good examples of this kind of system.
- *User-mode threads in a single process.* This class of threads has a library layer that emulates a multi-thread environment within a single process without any special kernel support. In this environment, a blocking system call such as an I/O request may or may not block all the process' threads depending on the features of the emulation library. Examples of this approach include Sun's lightweight process library [9], and DEC SRC's Topaz [10] Modula-2+ environment for ULTRIX.
- *Language supported.* Some systems provide a language interface for multi-threading. This is a cut above just library support, although the amount of support given by the language varies greatly. Examples of languages designed or extended for multi-threading include Ada, DEC SRC's Topaz Modula-2+ environment, Xerox's Mesa language [11], CMU's C Threads [12, 13], and Washington's C extensions [14].

Real systems often incorporate several of the characteristics given above but are not intimately tied to a particular implementation. For example, Taos uses threads for kernel mechanisms (threads in the kernel) and for user tasks (kernel-supported user threads). Topaz is layered upon Taos to provide a Modula-2+ multi-thread environment. However, Topaz is not tied specifically to Taos because Topaz can run on generic ULTRIX systems.

In designing ULTRIX Threads, a number of real-world constraints influenced our interpretation of threads. The following goals and non-goals helped focus our efforts.

- *Compatible.* We view ULTRIX Threads as an evolutionary step, and an important component of this evolution is to maintain some compatibility with the original system. We wanted to maintain the process model, and preserve kernel and library interfaces.
- *Balanced.* Threads are intended to be as lightweight as possible, making them suitable for automatic (compiler generated) or manual decomposition. Heavyweight mechanisms such as separate and protected address spaces, and separate sets of operating system objects (e.g., file descriptors) are not appropriate for threads. Threads balance light weight with requirements for thread autonomy such as independent scheduling and concurrent execution.
- *Minimal.* We sought to balance the split between kernel and library implementation so that kernel modifications are minimized but the system as a whole provides the functionality desired.

hardware dependent code. Details of whether there is a "core kernel" with one or more operating system interfaces [1] are beyond the scope of this paper.

[2] A "kernel thread" here refers to a thread in the kernel whose purpose is to run a user thread.

- *Portable.* The application interface for multi-threading should be portable to other systems. Special compiler, linker, or operating system support should not be assumed.
- *Simple.* The application of primitives provided at each level of the system should be obvious and straightforward. The intent is to provide simple primitives on which more complex or flexible mechanisms can be built.
- *No multi-threaded kernel.* Reimplementing the kernel to be multi-threaded, as opposed to its current monolithic nature, is beyond the scope of this project.
- *No thread visibility outside the process.* Unlike Mach's threads, ULTRIX Threads are not externally visible. For example, process A cannot signal a particular thread in process B; instead, the signal is sent to process B and handled by some thread in process B.
- *No auto-magic libraries.* Automatic detection or correction for the use of single-threaded libraries, particularly for third-party binary libraries, is beyond the scope of this project.
- *No multiple library interfaces.* We decided to adhere to compatible interfaces and not clutter the environment with alternate library interfaces.

The result, a prototype ULTRIX Threads, is an environment for multi-thread application development consisting of a ''kernel-supported user thread'' kernel, a multi-thread library, and an enhanced *dbx(1)* debugger.

The rest of this paper describes ULTRIX Threads in more detail, beginning with a discussion of the tradeoffs and decisions leading up to our final threads design, and the primitives that the kernel provides for building a multi-thread environment. This is followed by a description of the multi-thread library interface and functionality, and how it was used to convert libraries written for a single-threaded environment into libraries for a multi-threaded environment. Finally, the changes made to dbx to support multi-thread debugging, and the extensions to the command set for examining a multi-thread program are outlined.

2. Evolution of ULTRIX Threads.

The design of ULTRIX Threads evolved over time, during which a number of options were laid to rest. Early on, the idea of threads in the kernel was rejected because it was peripheral to our goal of supporting application threads. The remaining options were considered, and the reasons for selecting or rejecting various features are detailed below.

2.1. User-level Threads in a Single Process.

User-level threads in a single process were rejected because they are essentially coroutines that require the clients to schedule threads explicitly or depend on awkward mechanisms to simulate preemptive scheduling. Also, they cannot execute concurrently. We wanted ULTRIX Threads to be independent entities in the kernel, allowing the kernel to schedule them for concurrent execution in a multi-processor environment.

2.2. User-level Threads with Virtual Processors.

A virtual processors scheme was considered. Virtual processors are a two-level scheme where a small pool of kernel processes or threads support a larger number of user-level threads. The result is true thread semantics at the user level. Dynix and PCR use two-level schemes for their threads.

Virtual processors were rejected because of the difficulty in scheduling user threads on top of kernel processes or threads. How should user threads scheduling be coordinated with kernel threads scheduling?

2.3. Basic Kernel Support for Threads.

Gradually, our design moved towards a single ULTRIX process containing multiple threads. This required more kernel changes, but allowed for better thread semantics and control. The change entails splitting the process' schedulable entities from the shared information in the the *proc* and *u* structures.

An interim proposal of having only one master thread in a process able to make system calls was considered but rejected, since it would be awkward to have special threads with these capabilities, and it would be at odds with a symmetric multi-processing kernel.

The final ULTRIX Threads design is described below. We leaned towards simplicity in design, and left details and policy up to the multi-thread library.

3. ULTRIX Kernel Threads.

A thread is a flow of control within a process. All threads in a process share one set of some process information. A thread is scheduled independently, has it own stack, and can make its own system calls. Other resources, such as virtual memory and file descriptors, are shared between threads in a process. Threads are identified with 32-bit thread identifiers (TID).

3.1. System Calls.

There are new system calls to manipulate threads. We did not modify existing system calls to support threads.

```
int tfork(stack_ptr, prot_ptr)
    caddr_t stack_ptr, prot_ptr;
```

This call creates a new thread in the current process. The address space is shared among threads in the process. The two arguments are optional. The arguments point to the thread's stack and guard page that is read and write disabled. They are used by the multi-thread library so that the new thread has a stack as soon as it runs. The arguments are not retained in any kernel structure, and the kernel remains ignorant of user stack information.

```
int gettid()
```

This call returns the 32-bit thread identifier of the current thread.

```
void texit()
```

This call exits the current thread as ell as the process if there are no remaining threads.

```
killt(tid, signal)
```

This call sends a signal to a specific thread. The target thread must be a member of the same process as the signalling thread.

3.2. Tradeoff Issues.

An important design decision was to determine where to split the thread from the process. The less context a thread carries, the lighter weight it will be[3]; thus a goal was to leave as much as possible in the process. A hierarchy of information was created.

[3] The overhead for ULTRIX Threads was about 300 bytes of storage for inactive threads and 14 Kbytes of storage for active threads.

User:
  User ID, file system quotas, quotas of processes and threads.

Process group:
  Process group ID, target of killpg signals.

Process:
  Process ID, and shared resource information.

Thread:
  Control and scheduling, debugging, stack and signal information.

3.3. Virtual Memory.

All threads in a process have the same view of memory, which means that there is no thread-private memory. The threads share the page table entries associated with the process. This common view caused a problem because there was no memory location to store a thread's identity. We solved the problem by using an unused VAX processor internal register (ESP) to stored the thread identity. Another solution would be to use the thread's stack pointer as a guide to the thread identifier. This works because each thread uses a unique stack area.

3.4. Forking.

There was a question of whether a forking process should copy all the parent's threads. In our design, the new process contains only one thread that is a copy of the forking thread in the parent. This is because forks are generally performed either in preparation for an *exec(2)* or to gain parallelism. In both cases, it is sufficient to create a single thread in the forked process since:

1) processes that will immediately request an *exec(2)* do not need the other threads duplicated, and
2) most programs that want parallelism will fork threads rather than processes.

In either case, the other threads aren't necessary; thus it wasn't necessary to provide a multi-thread version of fork. This is the same approach taken by Mach.

3.5. Stacks.

Neither the ULTRIX kernel nor the ULTRIX Threads kernel know about managing user stack details. The user stack resides in user space as before, and is managed by the multi-thread library. The ULTRIX Threads kernel could allocate thread stacks, but it would need to know how to initialize and free the stacks, and how to interact with the user level stack allocator. This has not been the responsibility of the kernel and would add considerable complexity.

3.6. Signals.

One of the goals in the ULTRIX Threads design was to support old signal mechanisms. It was necessary to determine which entity receives which signals. An analogy exists in a ULTRIX system where a signal sent to a process group is ultimately received by individual processes. To determine what should happen for ULTRIX Threads, we examined how signals are currently used. Some signals are sent from outside the process, such as SIGTSTP or SIGINT. Others such as SIGSYS or SIGTRAP are generated by the kernel in response to user actions. Thus, signals are split into two classes. Signals sent from outside a process shouldn't know about the threads inside the process, so those signals continue to be sent to the process by the kernel using *kill(2)*. Those signals are ultimately delivered to each thread in the process. This apparently expensive operation is reduced via the *sigwait(2)* mechanism described below. Other signals are directed towards threads, so they are sent using *killt(2)*, a new system call. Clients are expected, but are not required, to follow this convention for uniformity. Signal handlers are associated with the process as a whole, but a specific thread handles the signal. This approach differs from Mach where the operating system chooses a randomly selected thread to handle signals [13].

3.7. Synchrony versus Asynchrony.

Signals are often used to notify processes of asynchronous events in a standard ULTRIX system. When an asynchronous event occurs, a signal is sent to a process that saves the current location, executes a special handler, and resumes execution at the saved location. This provides a primitive form of concurrency, and other operating systems use similar facilities. An example of this is the VMS Operating System's Asynchronous System Trap (AST). We no longer need to use this mechanism in a multi-thread environment. A thread can be dedicated to wait for an event instead of using a signal to notify processes of asynchronous events. We can afford to do this because threads are inexpensive, and since threads are dedicated to these tasks, they do not interfere with the state of the main program. Such a scheme allows cleaner and more elegant code. Threads now wait for these events through a new mechanism.

```
int sigwait(signals, command_flags)
    unsigned *signals;
    int command_flags;
```

Sigwait(2) is similar to *select(2)*. The *signals* parameter is a pointer to a signal mask to specify which signals the thread is waiting on. It is updated before returning to show which signals were received. The *command_flags* parameter specifies how the process' sigwait flags are cleared, or if the thread is to be suspended to wait for the bit to be set (polling). It is possible to dedicate more than one thread for an event, which blocks waiting for a signal to arrive and then performs its own handling code. The important point here is that *sigwait(2)* allows programs to catch **asynchronous events synchronously**. Pending signals are saved until the next *sigwait(2)* call. Similar issues were addressed in the Topaz programming environment.

3.8. SIGALERT - a New Signal.

A special signal is provided for the multi-thread library to use for cross-thread alerts. Its purpose is to interrupt threads in the middle of system calls, as well as tap another thread on its shoulder to catch its attention. It is an asynchronous signal and is accessed through the *killt(2)* syscall.

3.9. Scheduling Priorities.

Two different priorities are kept by the ULTRIX Threads kernel. The first is the standard one based on CPU usage, and the second is a process relative priority for threads. The *getpriority(2)* and *setpriority(2)* system calls have been extended to allow thread priorities to be read and set.

4. Library Support and Primitives.

A Threads library provides an interface to the kernel's multi-thread facilities for applications to use. This support may be broken down into two areas: primitives for multi-threading, and versions of the standard libraries for use by multi-threaded applications (multi-thread-safe). We emphasize the former here, and mention the latter in the context of the operations supplied by the multi-thread library.

4.1. Creating and Terminating a Thread.

The kernel interface is too low-level to be used directly by applications. The role of the multi-thread library is to provide a higher-level interface that handles much of the bookkeeping for the application. Included in these chores are allocation of stack areas, creation of thread control blocks, and maintenance of "glue" on which much of the remaining functionality depend.

```
thd_handle_type thd_create(proc, attributes, data_len, data)
                                    /* returns an opaque type */
   int *proc();                     /* proc to execute in new thread */
   thd_attributes_type *attribute;  /* priority, stack/guard sizes, ... */
   int data_len;                    /* size in bytes of data arg. */
   caddr_t data;                    /* application-specific data arg. */

void thd_exit(status)
     int status;
```

Thd_create(3t) is used to create another active user thread in the current process. The newly activated user thread is associated with a kernel thread, thread stack and thread control block, and will execute procedure *proc* with arguments *data_len* and *data* in an environment customized by the *attributes* argument. The kernel thread is initially created by the system call *tfork(2)*, and later may come from a cache of previously used threads maintained by the multi-thread library.

The multi-thread library handles allocation of thread stacks from user memory. The stack is at least the size requested in the *attribute* argument, allowing applications to tailor the amount of memory reserved for stack use. A default stack size is used when the application does not specify a preferred stack size. In the VAX/ULTRIX V2.2 prototype implementation, thread stacks are allocated from the P1 segment in contiguous eight page "chunks" with the default stack size being two chunks.

Each thread stack is separated by one or more guard pages. Guard pages are used to assist in detecting stack overflow during program execution. Variable-sized guard pages allow one to optimize stack overflow checking to just cases where the stack frame size may be larger than the guard size. In all other cases, the read-and-write protected guard page(s) will cause the thread to fault should a stack overflow occur.

Threads are less expensive than processes, but are not so cheap as to invalidate the usefulness of caching [6]. One of the *attribute* options is a cache-on-exit selector which allows the application to cache three different thread resources when the thread terminates: thread control block and associated user stack area, thread-private memory, and the corresponding kernel thread. Thread-private memory (described below) is a facility provided mainly to realize our second goal of preserving current stateful interfaces found in the standard libraries. Thread-private memory allocated by the standard libraries may be cached to avoid reallocation and initialization on subsequent uses of the thread control block. The kernel thread and its kernel stack may also be cached. Note that the thread-private memory and the kernel thread cannot be cached without also caching the thread control block.

Thd_exit(3t) is similar to *exit(3)* in that the current thread is allowed to perform application-specific cleanup, then exit. During cleanup, the thread's unwind handlers [10] perform an orderly termination such as release locks or deallocate temporary dynamic storage. In this sense, *thd_exit(3t)* is to *texit(2)* as *exit(3)* is to *_exit(2)*.

4.2. Signals and Exceptions.

4.2.1. Interprocess Signals and Sigwait(2).

The recommended model for handling asynchronous Unix signals is to dedicate a thread to handle them synchronously using the *sigwait(2)* mechanism, as described above. This thread then synchronizes with its cooperating threads to respond appropriately to the signal. The multi-thread library imposes further restrictions on applications by not allowing an application to install signal handlers except for synchronous signals (e.g., SIGFPE). Applications that make use of this "loop-hole" must be well-behaved while executing in the signal handler. For example, the handler must be careful when accessing shared data protected by certain kinds of synchronization mechanisms because that may result in deadlock.

4.2.2. Alerts: Cross-Thread Exceptions.

```
/* send alert to target thread */
int thd_alert(target)
    thd_handle_type target;
```

Occasionally, one thread will want to "poke another thread" to get its attention in a manner similar to the way processes use signals to interrupt each other. We use the kernel's *killt(2)* to implement this "poke," or *alert*. Alerts are based on a termination model of cross-thread exceptions, and force the target thread to execute *thd_exit(3t)* as soon as possible [10, 15]. Although there is no way to enforce the termination model, attempts by an application to implement a continuation model are discouraged because alerts are discarded while the thread is unwinding so as not to "terminate while terminating."

4.2.3. Controlling Receipt of Alerts.

We provide primitives to protect critical code regions which defer and re-enable alerts. A pending alert is taken when alerts are re-enabled. The default is to allow alerts. The application should provide unwind handlers to perform any necessary cleanup actions on termination.

4.3. Mutual Exclusion and Synchronization.

The most basic mechanism for handling concurrency provided by the multi-thread library is the mutex [12]. Mutexes are intended to protect short, well-behaved, critical code regions that are guaranteed to complete in a timely fashion. As such, while a mutex lock is held asynchronous alerts are automatically deferred.

In addition, we provide several other general-purpose synchronization mechanisms: condition variables (built using mutexes), readers/writer and a simple count-down semaphore (built using condition variables). Each one of these synchronization mechanisms was used in converting the standard libraries to be multi-thread-safe (e.g., readers/writer was used to control access to a shared hash table in *hsearch(3)*). These are the recommended synchronization primitives for general application use because they are also "alert-safe" in that locks will be released during the unwind phase of termination.

4.4. Thread-Private Memory Allocation.

Many of the standard library interfaces are "stateful." The two main flavors of this can be seen in *gethostent(3n)* and *crypt(3)*. In the former, a file pointer is maintained internally so that *gethostent(3n)* will return the next record in /etc/hosts. Also, *gethostent(3n)* returns its result in a static buffer — a mechanism incompatible with multi-threading [13].

In the case of *crypt(3)*, a set of routines share internal state (e.g., the key schedule used to transform input data into encrypted output) and this state may be maintained over multiple applications of a particular key schedule on input data.

For these cases, we simulate a single-threaded environment by allocating and maintaining thread-private memory at the library level to "unshare" what would otherwise be shared storage. This removes the shared storage synchronization requirements between threads. Note that this approach does prevent, for example, a single key schedule to be shared amongst all interested threads. The general case of designing interfaces for sharing is more complicated. The approach taken here is simple, and works well for this particular application.

```
caddr_t thd_mem_alloc(length, id, init_value, init_length)
        unsigned int length;       /* buffer size desired, in bytes */
        void (*id)();              /* unique tag for buffer */
        unsigned int init_length;  /* significant bytes in init_value */
        caddr_t init_value;        /* src of initial value(s) for buffer */


void thd_mem_free(buffer)
     caddr_t buffer;
```

Thd_mem_alloc(3t) allocates *length* bytes tagged with *id* (and implicitly with the TID), and whose first *init_length* bytes are initialized from *init_value*. If an existing buffer tagged with *id* is found, a pointer to it is returned. It is an error to change the length of an existing buffer. *Id* is a pointer to a ''destroy'' procedure, a convenient unique key, which is called when deallocating the buffer to cleanup the contents of the buffer (e.g., close file descriptors or deallocate secondary storage).

4.5. Errno and Single-Threaded Libraries.

The error status variable *errno* is handled as a special case of thread-private memory. Each thread accesses its own *errno* to avoid confusion [9].

```
#define errno thd_err_no()    /* map errno using a macro */
```

However, old single-threaded binary libraries may insist on accessing the old global status variable. Calls to such routines need process serialization as well as *errno* mapping between the thread-specific variable and the global variable. These functions are implemented by the following operations, which are placed around the call to the single-threaded library routines.

```
void thd_begin_dd();           /* dd = ''dusty deck'' */
     /* execute single-threaded code here */
void thd_end_dd();
```

5. Debugging

Old debuggers and kernel debugging interfaces are not sufficient to debug multi-threaded applications since they do not support thread-specific features to examine and manipulate threads. We extended *ptrace(2)* to support per-thread debugging functions, and a new version of *dbx(1)* was written to take advantage of those facilities. We chose *dbx(1)* since it is a popular debugger, and chose to use the existing *dbx(1)* source as a starting point to get an implementation done quickly. The new *ptrace(2)* call includes commands to:

1) Get the default traced thread's thread id,
2) Read and write a thread's process control block,
3) Send a signal to a thread,
4) Single step a thread, and
5) Define a list of threads to be made runnable for future *ptrace(2)* commands.

Note that no *ptrace(2)* command was provided explicitly to list the currently active threads. Whenever a new thread is created as a result of a *tfork(2)* in the traced process, the kernel sends the new thread a SIGTRAP signal. The debugger is made aware of this signal through the *wait(2)* mechanism and notes the new thread to keep in its list of active threads. Whenever the debugger tries to list the threads in the process, it must read the *U* area of each thread to obtain the current PC of the thread. If that operation fails, it means that the thread no longer exists and that thread is deleted from the debugger's list of threads.

The new dbx includes commands to list the thread id's of known threads, to change the context to a particular thread for commands to operate on, and to manipulate and name the subset of threads to operate on. Sets of threads are given names and assigned a set of thread ID's as its members. There is a command to set the default thread set that is in effect. The default thread *set* is different from the default thread. The default thread set limits the threads that are runnable, and the default thread is the single thread that commands operate on. For example, the print command examines a default thread's state, but the resume command will only allow the default thread set members to run. The default thread is set automatically on a breakpoint to be the thread that caused the stop, but may be changed manually by the user. Of course, the meaning of the existing commands were extended. For example, when the process is stopped, *all threads* are stopped, and when the continue command is given, all threads are resumed, given the constraints of the thread set. But with the step command, only the current thread will single step. We decided not to allow a special command to let a single thread to run, but that is possible by limiting the subset of runnable threads by defining a thread set with only one member. Various commands that print status information were changed to also print the thread id.

In hindsight, it would have been better to restructure or rewrite the debugger rather than to modify the existing one to retrofit threads features. That is because there are so many assumptions about the

1) semantics of processes, and

2) the use of *ptrace(2)* and *wait(2)* system calls which made it hard to work around.

For example, any one of the debugged threads in a multithreaded program may have hit a breakpoint upon return from the *wait(2)* system call. More bookkeeping needs to be done in the debugger to manage the threads, and fitting that into the old dbx structure was more cumbersome than expected.

Compatibility was important in the design of the kernel debugging facilities since we wanted to allow old debuggers to be run after recompilation. Although thread specific commands cannot be issued in such old debuggers, it will be possible to do limited thread debugging since you can examine the thread that caused a breakpoint or other stopping condition. You simply cannot change the thread context to examine another thread. The core dump format was also kept backwards compatible to allow older programs to examine it. The state info of the thread that caused the core dump is made available in the file location where the normal non-thread core dump would place it. The core dump also includes state info for all the other threads, but they are kept in a separate area that is not examined by older debuggers.

6. Conclusions

Building an ULTRIX Threads prototype environment confirms the importance and need for changes to each part of the system: the kernel provides basic support for multi-threading and multi-processing; the libraries provide an application interface, concurrency control, and run-time support; tools like *dbx(1)* support application development.

Although ULTRIX Threads involved widespread changes, the result is a system that still feels like UNIX, and is an extension of UNIX paradigms. The impact of the changes is quite manageable, as the following summarizes.

- *Simple*, not complicated. The multi-thread library does not provide a grab-bag of primitives, and it does not exude a multitude of programming paradigms. The model is simple: the multi-thread library will guarantee proper internal synchronization, and the familiar standard library interfaces are preserved by a multi-thread-safe implementation.

- *Compatible*, not incompatible. The kernel preserves the UNIX process interface, and provides a compatible process core dump. The multi-thread library preserves the standard library interfaces. The dbx debugger preserves the familiar command set.
- *Extension*, not replacement. The kernel provides threads in a process, not threads with a "process wrapper"; signals to a process group are delivered to each process, so signals to a process are delivered to each thread; inter-process signals have their parallel in inter-thread signals. The dbx debugger has an extended command set for dealing with multi-thread applications.
- *Portable*, not proprietary. The multi-thread library does not depend on special compiler or linker support. The primitives it provides can be implemented on any UNIX system, with or without kernel support.

Finally, we were able to meet most of our goals with just a few exceptions in the kernel and multi-thread library. One new header file and one new interface were added, and the *ptrace(2)* syscall was extended. The main departure from standard UNIX is the multi-thread library's support for handling signals. The handling of asynchronous events in a synchronous manner is one of the basic problems addressed by threads [10]. We do not allow applications to install asynchronous signal handlers; instead, we encourage application programmers to use the *sigwait(2)* mechanism.

7. Acknowledgements.

We acknowledge the following people and groups for their assistance in this project: Graham Hamilton, ULTRIX Threads project leader and kernel implementor, and Miche Baker-Harvey, ULTRIX Engineering Group; ULTRIX Engineering Group, ULTRIX consulting and design reviews; Garrett Swart and Paul McJones, Systems Research Center; Andrew Black, Distributed Systems A/D.

References

1. R.F. Rashid, "Threads of a new system," *Unix Review*, pp. 37-49, August 1986. Overview paper on Mach, with mention of threads on page 40.
2. D.D. Redell, Y.K. Dalal, T.R. Horsley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, and S.C. Purcell, "Pilot: An Operating System for a Personal Computer," *Comm. ACM*, vol. 23, no. 2, pp. 81-92.
3. D.R. Cheriton, "The V Kernel: A software base for distributed systems," *IEEE Software*, vol. 1, no. 2, pp. 19-42, April 1984.
4. L. Cooprider, R. Gurwitz, and T. Teixeira, "Support for tightly coupled processors," *Unix Review*, pp. 71-75, May 1988. An overview paper with some discussion of threads in Stellix.
5. J.M. Barton and J.C. Wagner, "Beyond Threads: Resource Sharing in Unix," *Winter Usenix Conference Proceedings*, pp. 259-266, Usenix Association, 1988. Detailed paper that describes systems implementation of process share groups, a mechanism similar to threads.
6. B. Beck and Dave Olien, "A Parallel Programming Process Model," *Winter Usenix Conference Proceedings*, pp. 84-102, Usenix Association, 1987. Sequent's system for parallel programming rather than threads, but talks of similar issues.
7. M. Weiser and A. Demers, "PCR: PARC Common Runtime," in *Bay Area Systems Seminar*, Palo Alto, October 1988. BASS Seminar presentation on Xerox PARC's portable runtime that includes threads.

8. T.W. Doeppner, Jr., ''A Threads Tutorial,'' Technical Report CS-87-06, Brown University, March 1987. User level threads.
9. J.H. Kepecs, ''Lightweight Processes for Unix: Implementation and Applications,'' *Winter Usenix Conference Proceedings*, Usenix Association, 1985. Sun's user level threads library paper.
10. P. McJones and G. Swart, ''Evolving the UNIX System Interface to Support Multithreaded Programs,'' *Winter Usenix Conference Proceedings*, Usenix Association, 1989. Describes Topaz and is also available as part 1 of DEC SRC Report 21.
11. B.W. Lampson and D.D. Redell, ''Experience with Processes and Monitors in Mesa,'' *Comm. ACM*, vol. 23, no. 2, pp. 105-117, February 1980.
12. E.C. Cooper and R.P. Draves, ''C Threads,'' Draft of Carnegie Mellon University Technical Report, March 1987.
13. A. Tevanian, ''Mach Threads and the Unix Kernel: The Battle for Control,'' *Summer Usenix Conference Proceedings*, Usenix Association, 1987. The Mach paper that concentrates on threads and many issues.
14. B. Binding, ''Cheap Concurrency in C,'' *SIGPLAN Notices*, vol. 20, no. 9, p. 21, September 1985.
15. T.W. Doeppner, Jr., ''Threads: A System for the Support of Concurrent Programming,'' Technical Report CS-87-11, Brown University, June 1987. User level threads.

NFSSTONE

A NETWORK FILE SERVER PERFORMANCE BENCHMARK

Barry Shein

Software Tool & Die

Mike Callahan

Paul Woodbury

Encore Computer Corporation

ABSTRACT

Network file servers are becoming a critical facility in modern computing environments. With the growth in their popularity and the emergence of multiple vendors providing software products which adhere to the same standards comes a need for relative performance measurement of different configurations. We have designed a benchmark and report our experiences with it on different configurations of servers and clients. The benchmark was designed to be portable (between networked file system protocols) and tunable to reflect different disk traffic patterns if desired. The default parameters used were chosen to be similar to the traffic patterns of typical networked file system environments as earlier reported in [SANDBERG85].

1. Introduction

Networked file servers, computers which provide their file systems remotely via standardized networking protocols, have become an important component in the design of any distributed computing environment. As a result of industry-wide standards systems designers find they have a choice of file servers all offering similar services.

Perhaps the most important requirement, beyond compatibility and adherence to standards, is performance. In one study of 363 users of Sun Microsystems products "75% of responders stated 'Occasional' or 'Frequent' performance limitations" as a concern at their site [EPOCH88]. The study reported that approximately 92% of their workstations were either diskless or dataless[1] implying critical dependence on a file server.

File servers have several performance components: Disk I/O, Network I/O, file cache efficiency, memory availability and CPU speed. Any one of these might affect file server performance. Thus, one can ask two questions, what is the overall performance of a particular file server and what component of the file server/client is the major bottleneck on a specific configuration. The benchmark we present in this paper can be useful in answering both questions, either by comparing file servers with similar configurations, or one file server as its configuration is varied.

[1] A 'dataless' workstation has a disk which provides local swapping/paging partitions and basic system files (boot, binaries, etc.) User files are kept on a remote file system.

2. Benchmark Design

2.1. General Assumptions

A file server's performance is measured at the client. A benchmark should stress the server from a client or clients in a manner which is reasonably representative of the kind of load seen during very heavy usage.

File servers make their file systems available to clients by satisfying several types of requests. These include reading data, writing data, looking up files and returning file status. This is not very different from the requests a local time-sharing user makes on a local file system except that the requests have been indirected through a network and some layers of protocol. In the case of Sun's NFS[2], for example, requests pass through NFS, RPC (remote procedure call), XDR (external data representation), UDP/IP and an ethernet[3] (and/or other media), in addition to the normal file system mechanisms of the server.

By designing an overall throughput oriented benchmark we have purposely avoided detailed analysis of individual layers of the networked file system and have tried to remain independent of the underlying protocols in use. This is accomplished by using a 'black-box' approach to the problem which measures the time required to perform a mix of high-level operations.

This benchmark should be as useful in comparing different networked file systems (e.g., NFS vs. RFS[4]) as it is in comparing different implementations, both hardware and software, of the same networked file system.

2.2. Definition of NFSSTONE[5]

We have called the units reported by the benchmark **NFSSTONEs.** This is the total number of operations in one run through the program (45,522 with the default parameters used throughout this paper) divided by the total elapsed time in seconds. This can also be thought of as NFS operations per second, where NFS operations represent a mixture of requests tuned to reflect what we believe is normal usage (although compressed into a small time.)

2.3. NFS Operations Mixture

Since the mix of operations can be critical the benchmark program was designed to allow the person performing the tests to easily vary the proportions of reads, writes, lookups etc. The default values, however, were based on a particular mixture obtained by empirical measurement by [SANDBERG85] as follows:

NFS Operation	Sun %	NFSSTONE %
lookup	50	53.0
read	30	32.0
readlink	7	7.5
getattr	5	2.3
write	3	3.2
create	1	1.4

Sun's statistics were determined by compiling nfsstat statistics, our numbers were obtained empirically by observing similar kernel meters after a single run of our benchmark. In short, the operation mixture is purposely very similar. We considered the differences minor so did not attempt to juggle the benhchmark further to fit exactly. It would be more productive to repeat the Sun measurements in various environments to further refine these numbers.

[2] NFS is a trademark of Sun Microsystems, Inc.

[3] Ethernet is a registered trademark of Xerox Corporation.

[4] RFS is a registered trademark of AT&T.

[5] The suffix "stone" derives from a tradition going back to the Whetstone benchmark, dhrystone, dhampstone etc. Unfortunately we couldn't think of any hygroscopic prefix nor mean to imply any networked file system as such.

We consider this choice of default parameters to be sufficiently compelling to recommend them strongly when comparing file servers. We are happy that, should different parameters be desired, it is easy to vary the benchmark's values to fit new models.

A theoretical upper bound of throughput can be calculated for our particular mixture. If one assumes 100% utilization of a 10Mb ethernet (different media would be adjusted accordingly) and adds up all the data which is passed between client and server and, finally, assumes that pure protocol overhead and other operations such as lookup costs nothing, then we arrive at an upper bound of 421.5 nfsstones[6]. This might be thought of as a protocol-independent upper bound (no protocol, no matter how lightweight, should be able to exceed this.)

If one adds a cost for the transactions by assuming the average round-trip cost of an operation is 256 bytes (arbitrary, reasonable sounding number), including protocol headers, then the upper limit drops to around 389 nfsstones. This means that if a reported figure exceeds 400 nfsstones we would suspect that either caching is becoming significant or something is awry with the way the benchmark was being run.

2.4. File System Caches

The file cache in the client is a factor which can critically affect the performance of the overall networked file system. Caches are typically allocated at bootstrap time to be a percentage of available main memory or, in any case, will tend to reflect the total amount of memory in the client.

We have decided to try to mix read strategies which both uses the cache and defeats it. The assumption is that simple sequential reading of files promotes efficient use of the cache while 'randomly' reading blocks in large files will tend to defeat the cache (i.e., minimize cache hits.) This is based on experience with buffer cache designs and could be refined by actual measurement.

This can be contrasted with [KRIDLE83] where the motivation was to allow the cache to operate by allowing locality of file reference. By stressing the cache we get a better feel for the capability of the other system components, achieve some independence from particular client memory configuration and obtain results better resembling the behavior of servers with large numbers of clients. Clients with very large caches might force some rethinking of the default parameters. In [RODRIGUEZ88] a design is described which allows dynamic resizing of buffer caches and other kernel resources. On such systems attempting to defeat the cache further might distract from a fair measurement of what the system is trying to provide.

We perform other operations in a straightforward manner, creating directories and files and performing down-scaled versions of the basic read/write tests to these new files. A list of the specific operations can be found in appendix 1.

2.5. Synchronization of Clients

We use the term *synchronization* to mean starting the benchmark on more than one client at the same time. If the clients are reasonably synchronized then we expect a lot of overlap among clients.

To run the benchmark on several workstations we use a simple synchronization mechanism so all tasks start roughly simultaneously. This is accomplished by using a simple control program which will create a file, use the file system mechanisms to lock it and then start up the clients. The control program waits a short time (the clients all block waiting for a lock on the same file) and then unlocks the file allowing each client's lock request to complete and the benchmark to begin. Other synchronization mechanisms could be devised (e.g., if a system did not support network lock semantics) with similar effect.

Although synchronization is not essential, we desire a lot of overlap from the workstations. We have designed the benchmark to run a sufficiently long time on a typical system so that synchronization within several seconds is more than adequate to guarantee the desired concurrency. Our benchmark typically takes from several minutes to nearly an hour to complete. The time depends on the number of clients and other factors such as the hardware and software configurations available on the server.

[6] More precisely, 45,522/(135MB / 1.25 MB/s), where 45,522 is the total (default) NFS operation count as described earlier in the paper. Size of reads and writes are summarized in the appendix.

2.6. Algorithm Overview

After synchronizing on the lock file the time is noted. We then create a directory and fork children until 6 child processes are running. Each child creates a file, writes to it, and reads from it in two patterns: Sequential and non-sequential. The file is treated as a group of fixed sized blocks. These blocks, when read non-sequentially, are chosen by seeking to block offsets *(1, n, 2, n-1, 3, n-2, ...)* consecutively[7]. Mixed in with these are other operations such as creating and reading back symlinks, renames, creates, mkdirs and deletes. The exact sequence is summarized in appendix 1.

2.7. Coding

The benchmark is coded in C. Although we do not use the stdio library (to help avoid any differences in implementation) we read and write in units which should be similar to those chosen by stdio with block buffering.

2.8. The NFSSTONE

The result of the benchmark is a single number which we call an NFSSTONE. We felt it was important to be able to summarize results as one, essentially unitless, number so different experiments could be easily and quickly compared.

3. Test Configurations

To produce some sample results the benchmark was run on the following configurations:

3.1. Servers

Our servers can be characterized as modern, virtual-memory microprocessor-based systems with relatively fast disks and disk channels. Our server is a symmetrical multi-processor system using shared memory so we vary the number of CPUs as an additional variable.

3.2. Clients

The clients were all Sun3/60 workstations with either 8 or 12MB of memory running SunOS[8] release 3.4. Some were diskless and others dataless, the program itself is small (45KB total) and resides on another system not being tested. The clients are otherwise quiescent and had sufficent memory, so there is no reason to believe there is any significant interaction with paging I/O or other factors.

4. Results

All results are reported in units of *NFSSTONEs*. Our program runs six tasks (forks) on each client. We report results based upon the total number of tasks; to calculate the number of clients involved (i.e., workstations) simply divide the number of tasks by six.

4.1. Varying The Number of Disks

In this first example we use a system with 4 CPUs, 64MB of memory and one disk. Running NFSSTONE from six to twenty-four tasks we measure a 20% decrease in total NFSSTONEs. When we split the clients across two disks we see some decrease in nfsstones as we increase tasks. The total throughput increases about 50% when compared with the one disk configuration. This would suggest that the first configuration was disk limited.

[7] We are certainly open to suggestions for other block choice algorithms and reasoned arguments justifying those choices. One suggestion was to try a quadratic hash modulo some large prime as a block choice algorithm. This and other methods should be tried in the future for comparison, particularly with configuration variations such as striped file systems.

[8] SunOS is a trademark of Sun Microsystems, Inc.

64MB, 1 Disk Channel, 4 CPUs				
	Tasks			
Disks	6	12	18	24
1	85	79	69	68
2		120	109	112

4.2. Varying the Number of CPUs

For this data we measure total NFSSTONE throughput while varying the number of CPUs (each CPU is rated about 2 MIPs.) We see some increase between four and six CPUs but adding two more for a total of eight shows very little improvement.

64MB, 4 Disks, 2 Disk Channels				
	Tasks			
CPUs	6	12	18	24
4	83	133	153	158
6	85	147	167	183
8	85	150	179	187

The asymptotic increase from left to right in each row would indicate that the server is not yet saturated with fewer tasks.

4.3. Varying the number of Disk Channels

Comparing one and two disk channel configurations we see some difference as the number of clients increases. This suggests that simply splitting clients among disks can be further augmented by splitting the disks themselves among channels although not as dramatically as adding disks or CPUs.

64MB, 4 Disks, 4 CPUs				
	Tasks			
Channels	6	12	18	24
1	82	125	139	147
2	83	133	153	158

4.4. Conclusions

We believe that the benchmark is sensitive to variables that both customers and vendors wish to see measured. Customers want to know where to best spend their hardware dollars and vendors want to know how to improve their product. Both want to know where particular systems stand against each other. Benchmark results allow them to examine price/performance differences.

It would be nice if we could report how many nfsstones per client are 'necessary' but that varies from application to application. What we have measured can be used to compare different hardware and software implementations.

We plan to continue our experiments with this benchmark. Such experiments will include many more clients, different numbers of tasks on each client, multiple ethernets connected to one server and new network mediums as they become available.

The hope is that this paper catalyzes others to join us in refining an acceptable benchmark of networked file system performance and to define suitable methodologies for testing. We do not consider this work complete although we feel that it does set some framework for further development. We will make all our programs publicly available through the usual channels (anonymous ftp, usenet etc.) This should make results from many more systems and configurations available to the community.

The proportions of operations used for this paper were based upon the results in [SANDBERG85]. Repeating those measurements in carefully chosen environments would make those assumptions more rigorous.

Networked file system performance is important, but it is not the entire story. Two systems might perform similarly on this benchmark but one can be straining while the other may have quite a bit of capacity to spare, this should be the case when either the network bandwidth has been fully utilized or the clients become the limiting factor. Thought should be given to what services, beyond networked file systems, servers should provide and how this interacts with networked file service.

5. Acknowledgements

This NFSSTONE benchmark was inspired by earlier work done by Howard Eskin, Chris Jolly and Scott Palmer of the General Electric Corporate Research and Development Center. We are indebted to Howard Eskin for his many comments, criticisms and suggestions during the preparation of this paper, we hope to eventually produce a benchmark which meets his very high standards.

References

[EPOCH88] Epoch Systems Inc., "File Server Needs of High-Performance Workstation Users", June 1988.

[KRIDLE83] Kridle, B., McKusick, K., "Performance Effects of Disk Subsystem Choices for VAX Systems running 4.2BSD Unix", USENIX Summer '83 Conference Proceedings, pp. 156-169.

[MCKUSICK83] McKusick, M., Joy, W., Leffler, S. and Fabry, R. "A Fast File System for Unix", University of California at Berkeley, Computer Systems Research Group Technical Report #7, 1982.

[RODRIGUEZ88] Rodriguez, R., Koehler, M., Palmer, L., Palmer, R., "A Dynamic Unix Operating System", USENIX Summer '88 Conference Proceedings, pp. 305-319.

[SANDBERG85] Sandberg, R., "The Sun Network File System: Design, Implementation and Experience", Sun Technical Report. A version also appeared in the USENIX Summer 1985 Conference Proceedings, pp. 119-130, although not with the appendix of NFS operations we reference.

Appendix 1

Overview of NFSSTONE operations

```
lock/synchronize
note time
repeat 2
        mkdir
        repeat 3
                mkdir
                (child)
                create
                seq_write
                seq_read
                symlink
                repeat 583
                        readlink
                end
                nseq_read
                unlink
                repeat 83
                        create
                        fsync
                        close
                end
                unlink
                seq_read
                repeat 4167
                        access
                end
                seq_read
                nseq_read
                close
                rename
                unlink
                (end child)
        end
end
repeat 2
        repeat 3
                rmdir
        end
end
note time
report
```

Notes:

seq_write: sequential write of 250 * 8192 blocks.

seq_read: sequential read of 250 * 8192 blocks.

nseq_read: non-sequential read of 250 * 8192 blocks (see text for description.)

UFOS: An Intelligent Real-Time Performance Monitor for UNIX® System V

Charles Ballance (attmail!cballance)
Sean Fleming (attmail!sfleming)
Jay Goldberg (attmail!jgoldberg)
Nelly Karasik (attmail!nkarasik)

AT&T - Information Management Services
UNIX Technical Support
30 Knightsbridge Road
Piscataway, NJ 08854

ABSTRACT

The UNIX Facility for the Observation of Systems, UFOS, is an intelligent real-time performance monitor for systems running UNIX System V. UFOS monitors system-wide activity and alerts the user to potential performance problems, indicating what may be wrong and where to look for further information. UFOS is an interactive system designed to assist with the analysis of UNIX System V performance. The system can be customized to monitor the performance of a particular machine.

1. Introduction

In the beginning, UNIX systems were heavily used for documentation preparation and application development, the ratio of system programmers to systems was high, and system performance problems resulted in:

- A few users dissatisfied with the turnaround time for their *nroff* or *cc* commands.
- An aggressive lunch-time discussion among the system administrators regarding "those users who don't know good performance from bad."
- Late-night file system reorganizations and possibly even a *kernel hack* or two attempting to tweak some Murray Hill algorithm and prove to one's compatriots one's readiness for the next degree of *guruship*.

Rarely, if ever, did a corporation lose dollars over these performance problems.

Today, things have changed. UNIX systems are becoming a part of running the business. The proliferation of UNIX systems has reversed the ratio of system programmers to systems. Applications running under UNIX systems have become increasingly complex and in some cases may be running on extremely expensive hardware.

System performance issues are no longer simply a source for active lunch-time conversation. They need to be addressed and resolved before they result in loss of revenue. They need to be accessible to administrators with little understanding of UNIX system internals. Such requirements depict the need for *production quality* tools.

The screen modules can be navigated in a variety of ways. There are currently thirty-five modules in the system. The following illustration (Figure 1) delineates the hierarchical structure of the data display screens. Starting with Main and moving from left to right, levels of increasing detail are shown by the titles of the screens.

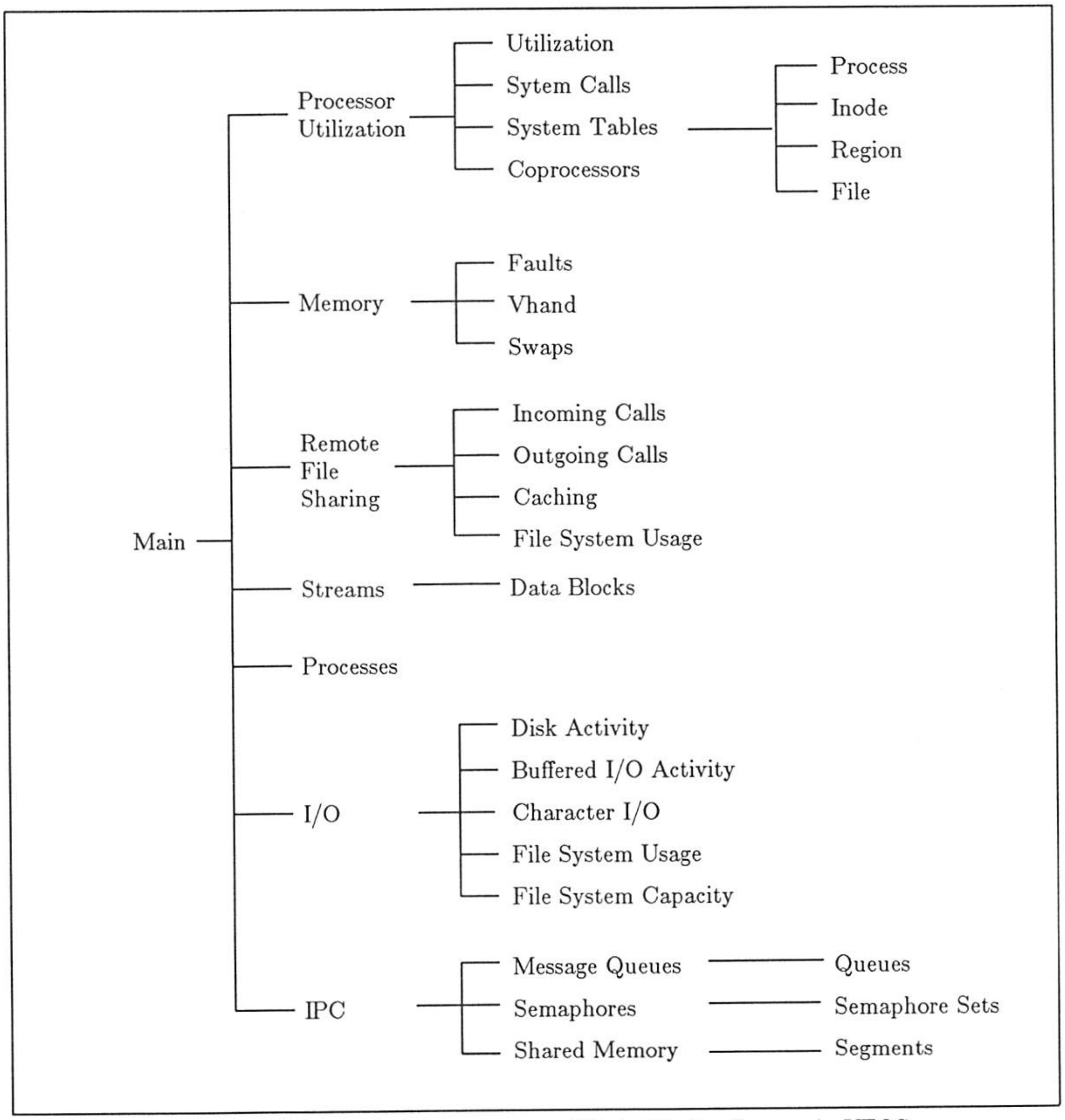


Figure 1. Hierarchical Arrangement of Data Display Screens in UFOS

2. Background

Performance analysis has traditionally been a difficult process for administrators of UNIX systems. Tools such as **sar**, the system activity reporter, have been the primary means of finding performance bottlenecks on UNIX System V systems. In order to identify a problem through the use of such a tool, a certain degree of expertise with system internals is required. As the UNIX system continues to mature and expand into new areas, it is becoming more and more difficult for an administrator to be well versed in all aspects of system performance. In addition, most software and hardware vendors do not provide comprehensive guidelines for system performance.

Thus, a need was found for an interactive performance monitor providing both the scope necessary for the observation of system-wide activity and the intelligence to aid the user in identifying problematic situations for UNIX system performance.

3. Overview

UFOS, pronounced "you-eff-owes", was conceived as an aide for investigating and analyzing problems associated with system performance. The package was developed for internal use within AT&T and, currently, it is not available outside of AT&T. The package is based on UNIX System V Release 3.

UFOS samples data structures populated by UNIX System V in the course of a machine's operation and presents such information interactively and dynamically. Related data items are grouped categorically and appear in full-screen format. UFOS makes decisions regarding the impact of observed activity on operation of the system, presenting items that may indicate performance difficulties highlighted (in inverse video), directing the user's attention immediately to the issues at hand.

The tool provides a default base of both rules and high(low)-water marks. The thresholds incorporated by UFOS for targeting performance problems are tunable by the user. If default values provided are found to be inappropriate, the guidelines used by the tool can be customized to best suit the needs of a specific system or application. The package can be used by the novice to locate problematic behavior. It can be enlisted by the expert to quickly locate areas of possible anomalous behavior.

4. Design Concepts

4.1 Hierarchical Structure

The UFOS user interface is composed of a hierarchy of screen modules. The hierarchical structure reflects a logical breakdown of the operating system and various extensions. The inverted tree structure is designed to guide the user from the broadest view of the system to more specific issues. At the broadest view, the following subsystems are monitored:

- Processor Utilization
- Memory
- Remote File Sharing
- Streams
- Processes
- Input/Output
- Inter-Process Communication

4.2 Analytic Capabilities

Upon entering the system, the main screen is presented. Whatever screen the user is interfacing with is the *current* screen. In the present release, only one screen may be presented at a time. On any particular screen, two types of analysis are performed. The first is the analysis of the displayed performance data values. Anything falling outside "default or user-tuned" thresholds will be highlighted. The second analysis is an examination of information associated with screen modules below the current screen in the hierarchy. Subscreens contain more specific information pertinent to the current screen. This submodular examination is done recursively down to the bottom of the tree structure. The analysis will result in the names of suspect subsystems being highlighted. From the main level of the hierarchy, all subsystems are checked by this feature for potential performance problems. This one-step comprehensive monitoring represents a departure from the repetitiveness inherent in the traditional performance monitoring tools.

Decisions about the traversal of the UFOS hierarchy and the identification of troublesome issues are automated by the intelligence feature of the package. UFOS incorporates high- and low-water marks as the basis for its decision-making processes; as the issues are better understood by the user, these values may be changed and saved so that acquired expertise is never wasted and a lesser amount of time is spent in performance troubleshooting. This gives the user the opportunity to correct past mistakes and to grow with knowledge gained.

The analysis of data is not simply confined to the checking of thresholds. The system also uses combinational logic to determine if data is worth investigation. This is not obvious to a user of the system. Although all thresholds are currently modifiable by the user, the base of rules is currently not available for manipulation.

The thresholds available with the system can be generally broken down into two types. Those that are hardware dependent and those that are defined independently of any particular hardware. For example, UFOS defaults the low water mark for free inodes in a file system to be ten percent. This default would reasonably apply to any hardware. On the other hand, the rate at which a system can perform system calls is something that has to be calibrated for each new type of hardware system.

Setting these types of thresholds was the most difficult part of developing the system. A base processor was benchmarked to perform various activities until certain levels of degradation were experienced. Thresholds derived for that system were then scaled to apply to other processors. This method was used for a portion of the hardware dependent thresholds. Published thresholds, general knowledge and common sense were used to set the remainder of thresholds. These rules of thumb are not cast in stone. As users relate their experiences with the product, the thresholds will undergo modifications.

In addition to data highlighting, specific items are explained via an on-line help facility. UFOS provides help for each screen module in the system, including general conceptual information and explanations of each data item presented. These explanations not only define the data items appearing, but they also detail their impact on system performance if targeted by the intelligence facility of UFOS.

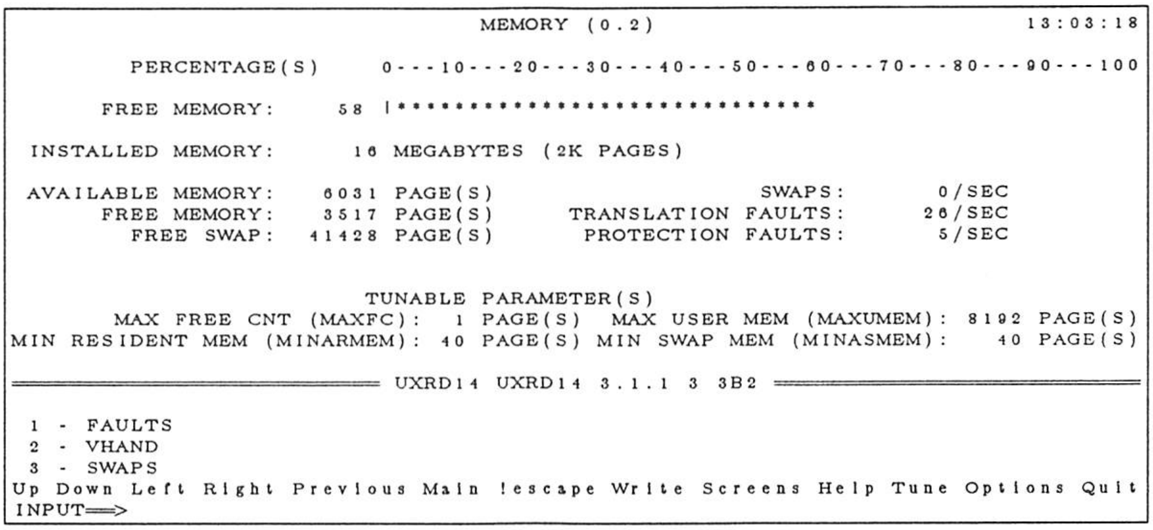
4.3 Data Display Screens

The fundamental unit of the UFOS user interface is the data display screen. These screens are currently managed via the **curses** facility. Thirty-five of these screens currently define the hierarchy incorporated by the tool.

Each data display screen makes use of a standard format containing the following features:

- Screen Title and Time
- Data Display Area (scrolling or static)
- System Name/UNIX System V Release Information
- List of Subscreens
- Command List
- Input Line

These screens are designed to aid the user; they present information relevant to the subsystem currently being investigated, making use of bar-charts and highlighting for expedient identification of items and issues. Helpful info is also incorporated where appropriate. Figure 2 is an example of a UFOS screen. The screen title is "Memory". Its path in the hierarchy is "0.2".



```
                                 MEMORY (0.2)                                  13:03:18

         PERCENTAGE(S)      0---10---20---30---40---50---60---70---80---90---100

       FREE MEMORY:      58 |******************************

 INSTALLED MEMORY:       16 MEGABYTES (2K PAGES)

 AVAILABLE MEMORY:     6031 PAGE(S)                 SWAPS:        0/SEC
       FREE MEMORY:    3517 PAGE(S)     TRANSLATION FAULTS:      26/SEC
         FREE SWAP:   41428 PAGE(S)      PROTECTION FAULTS:       5/SEC


                          TUNABLE PARAMETER(S)
        MAX FREE CNT (MAXFC):  1 PAGE(S)  MAX USER MEM (MAXUMEM): 8192 PAGE(S)
MIN RESIDENT MEM (MINARMEM): 40 PAGE(S) MIN SWAP MEM (MINASMEM):    40 PAGE(S)

=========================== UXRD14 UXRD14 3.1.1 3 3B2 ===========================

 1 - FAULTS
 2 - VHAND
 3 - SWAPS
Up Down Left Right Previous Main !escape Write Screens Help Tune Options Quit
INPUT==>
```

Figure 2. Example of Screen Format (Screen 0.2, Memory)

Beneath the screen title is the data display area. This is a mix of dynamic and tunable information. In the example, we see how much free memory is currently available on the system. We also see how much physical memory is installed on the system and a note about the size of memory pages. Available memory (non-kernel) is shown in terms of pages. Information on paging and swapping is also shown. Near the bottom of the data display area are a list of relevant tunable parameters. These will highlight along with other data values, if necessary.

Below the data display area is the system name and release information. Below that, we see the submodules available from this screen (i.e. Faults, Vhand and Swaps). The basic concepts of the hierarchical structure are available from the screens themselves. Screen 0, the main screen, is the initial (root) screen encountered by a user who invokes UFOS. Nodes in a path are delimited by periods. In this example, Screen 0.2 would be the second child screen under the main screen. In addition, the names of the subscreens of Screen 0.2 are given on the bottom of that screen. They are: Screen 0.2.1 (Faults), Screen 0.2.2 (Vhand), and Screen 0.2.3 (Swaps).

On the bottom of the screen, the available commands are displayed along with a line for user input and error messages. The commands *Up, Down, Left, Right, Previous* and *Main* are navigational commands for screen control. Full and relative pathnames of screen modules are also allowed. *!escape* is used to escape to a shell. *Write* copies the contents of the screen to a file. *Screens* gives a birds-eye view of the screen hierarchy. *Help* gives general information about the current screen, its data and tunables. *Tune* is used to modify the system thresholds for highlighting. *Options* is used to control how the system operates (i.e. time interval for updates). *Quit* exits a user from the system.

4.4 UFOS Data Collector

System activity is sampled according to a time interval that may be specified by the user for control over the granularity of the data. Relevant information is then checked by the intelligence facility of the tool to determine possible impact on system performance, and these data items are placed on the screen for the user, highlighted if they deserve notice. The rule of thumb is that a bigger interval provides a better sample for analysis and is also less burdensome on the system.

Data collection is the most intense activity handled by UFOS. The collected data can be generally broken down into two categories, rate and snapshot. Rate type information requires two samples to ascertain a value. Snapshot type info requires one. Several strategies are used to keep UFOS from becoming a performance problem in and of itself:

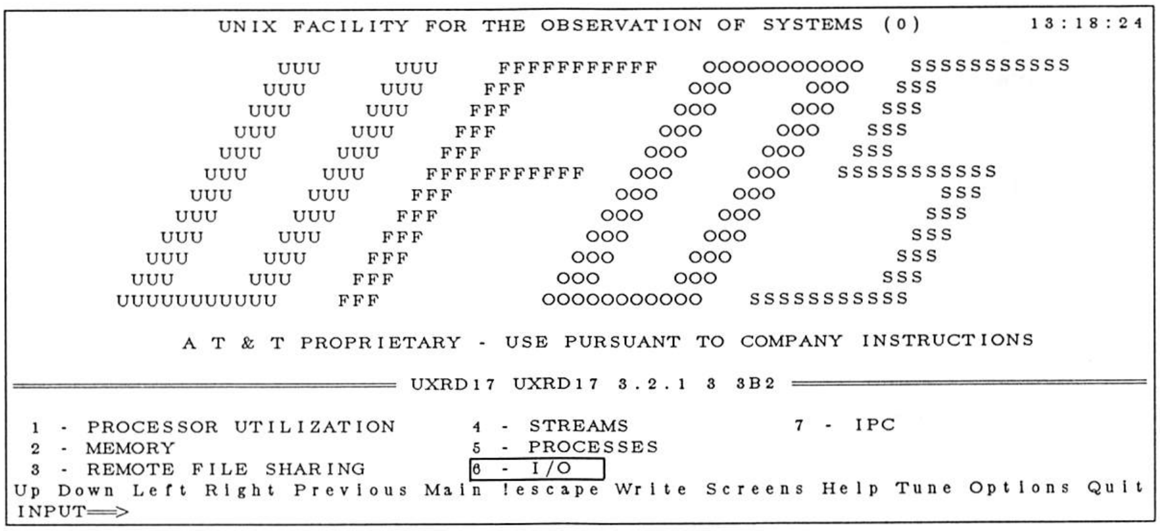
- Rate type information related to all modules is always collected at the expiration of a time interval. Snapshot type info is only collected for the current subsystems.
- The system performs its data processing using integer calculations.
- Some of the more intensive screen modules are not examined on every pass, but are staggered.
- When a subsystem is being recursively examined, the examination terminates when a problem is found.
- Addresses of system data structures are kept cached in a file in a manner similar to sar.
- A user can lessen the intensity of the system by enlarging the default time interval for screen refresh.
- The user has the ability to disable the recursive examination of the system. This lessons the intensity of the system but also lessons its effectiveness. The current screen is still evaluated.
- The scheduling priority (nice value) of UFOS is also controllable by the user. This can be increased to give it a better chance at executing on a busy system or decreased to give deference to other processes.

All in all, we have found that the package generally consumed between four and seven percent of the CPU on our various test systems.

5. Problem Solving Using UFOS - an Example

A system administrator receives a call from a programmer working on a database application, complaining that response time has degraded significantly. The administrator then invokes UFOS. Figure 3 depicts the Main screen of UFOS. Unlike other screens in the system, the main screen possesses no performance data values. It simply lists the available subsystems.

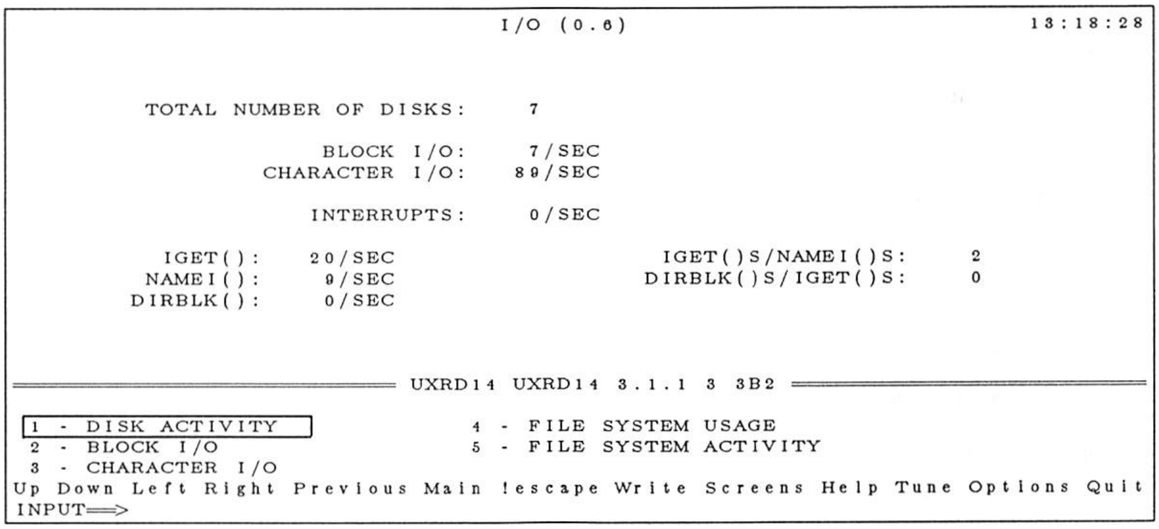
Rectangles surrounding data values or subscreens in the following illustrations are representative of screen highlighting.



```
              UNIX FACILITY FOR THE OBSERVATION OF SYSTEMS (0)             13:18:24
                     UUU      UUU     FFFFFFFFFFF   OOOOOOOOOOO    SSSSSSSSSSS
                    UUU      UUU     FFF           OOO      OOO   SSS
                   UUU      UUU     FFF           OOO      OOO   SSS
                  UUU      UUU     FFF           OOO      OOO   SSS
                 UUU      UUU     FFF           OOO      OOO   SSS
                UUU      UUU     FFFFFFFFFFF   OOO      OOO   SSSSSSSSSSS
               UUU      UUU     FFF           OOO      OOO           SSS
              UUU      UUU     FFF           OOO      OOO           SSS
             UUU      UUU     FFF           OOO      OOO           SSS
            UUU      UUU     FFF           OOO      OOO           SSS
           UUU      UUU     FFF           OOO      OOO           SSS
          UUUUUUUUUUUU     FFF           OOOOOOOOOOO   SSSSSSSSSSS

              A T & T PROPRIETARY - USE PURSUANT TO COMPANY INSTRUCTIONS
=============================== UXRD17 UXRD17 3.2.1 3 3B2 ==========================

 1 - PROCESSOR UTILIZATION       4 - STREAMS               7 - IPC
 2 - MEMORY                      5 - PROCESSES
 3 - REMOTE FILE SHARING         6 - I/O
Up Down Left Right Previous Main !escape Write Screens Help Tune Options Quit
INPUT==>
```

Figure 3. Main Screen with I/O Subscreen Highlighted

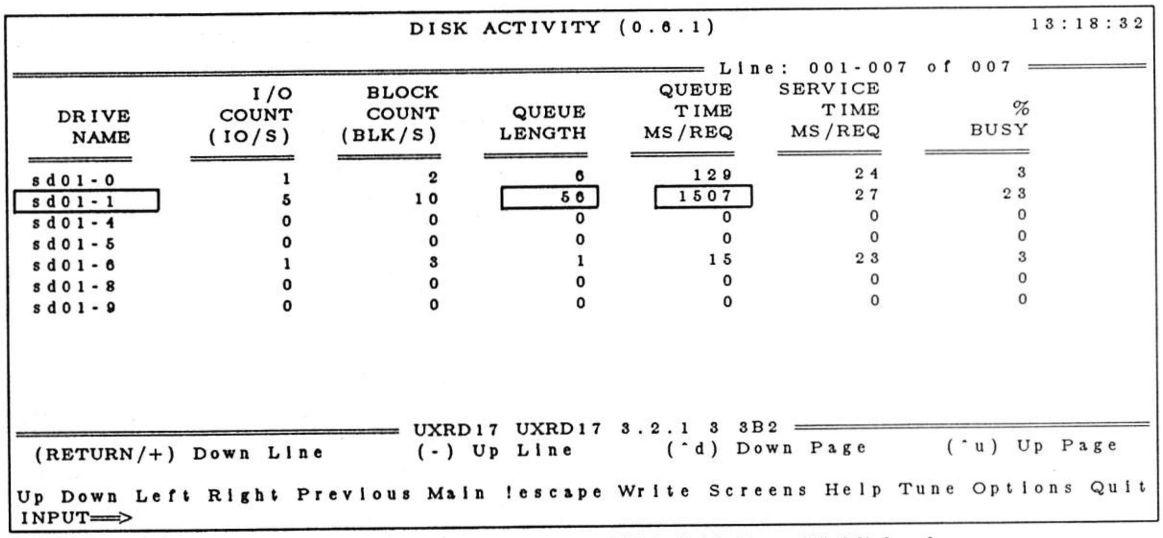
Seeing that the I/O subsystem (i.e., 6 - I/O) appears to be causing a problem, the administrator invokes the corresponding subscreen:



```
                                   I/O (0.6)                                13:18:28


         TOTAL NUMBER OF DISKS:       7

                    BLOCK I/O:        7/SEC
                CHARACTER I/O:       89/SEC

                   INTERRUPTS:        0/SEC

          IGET():      20/SEC                  IGET()S/NAMEI()S:      2
        NAMEI():        9/SEC                DIRBLK()S/IGET()S:      0
       DIRBLK():        0/SEC


=============================== UXRD14 UXRD14 3.1.1 3 3B2 ==========================

 1 - DISK ACTIVITY               4 - FILE SYSTEM USAGE
 2 - BLOCK I/O                   5 - FILE SYSTEM ACTIVITY
 3 - CHARACTER I/O
Up Down Left Right Previous Main !escape Write Screens Help Tune Options Quit
INPUT==>
```

Figure 4. I/O Screen with Disk Activity Subscreen Highlighted

This screen gives a general overview of I/O activity. Despite the low values of the data items appearing on this screen, the automatic scanning of subscreens performed by UFOS has turned up a problem below this screen. Following the prompting of UFOS, the administrator next invokes the screen for disk activity (1 - DISK ACTIVITY):



```
                              DISK ACTIVITY (0.6.1)                                13:18:32

=============================================== Line: 001-007 of 007 ==========
                 I/O        BLOCK                    QUEUE     SERVICE
  DRIVE         COUNT       COUNT       QUEUE        TIME       TIME          %
   NAME        (IO/S)      (BLK/S)      LENGTH      MS/REQ     MS/REQ        BUSY
 ========     ========    ========    ========    ========    ========    ========
 sd01-0            1           2           6         129         24           3
 sd01-1            5          10          56        1507         27          23
 sd01-4            0           0           0           0          0           0
 sd01-5            0           0           0           0          0           0
 sd01-6            1           3           1          15         23           3
 sd01-8            0           0           0           0          0           0
 sd01-9            0           0           0           0          0           0




========================== UXRD17 UXRD17 3.2.1 3 3B2 ===========================
 (RETURN/+) Down Line      (-) Up Line      (^d) Down Page      (^u) Up Page

Up Down Left Right Previous Main !escape Write Screens Help Tune Options Quit
INPUT==>
```

Figure 5. Disk Activity Screen With Disk Data Highlighted

The administrator sees the heavy activity on a single disk, its queue length and queue time indicating that this device is being given more requests than it can handle. In this case, the problem was that the two busiest file systems were separated on the disk unit by backup slices that were normally not mounted. This particular problem was corrected by placing the two file systems contiguously on the disk unit.

6. Customization

Commands are available from each of the data display screens within the UFOS hierarchy for the customization of the UFOS tool.

6.1 Run-time parameters

The following run-time parameters of the UFOS system are dynamic:

- Time Interval
- Recursive Submodular Examinations
- Running Priority (or *nice* value)

Changing the time interval allows the user to control the granularity of the data presented by UFOS. Enabling the submodular examination feature gives UFOS full use of its analytic capability. Changing the running priority of UFOS allows the user to exercise more control over its ability or need to be scheduled to run on an overutilized system.

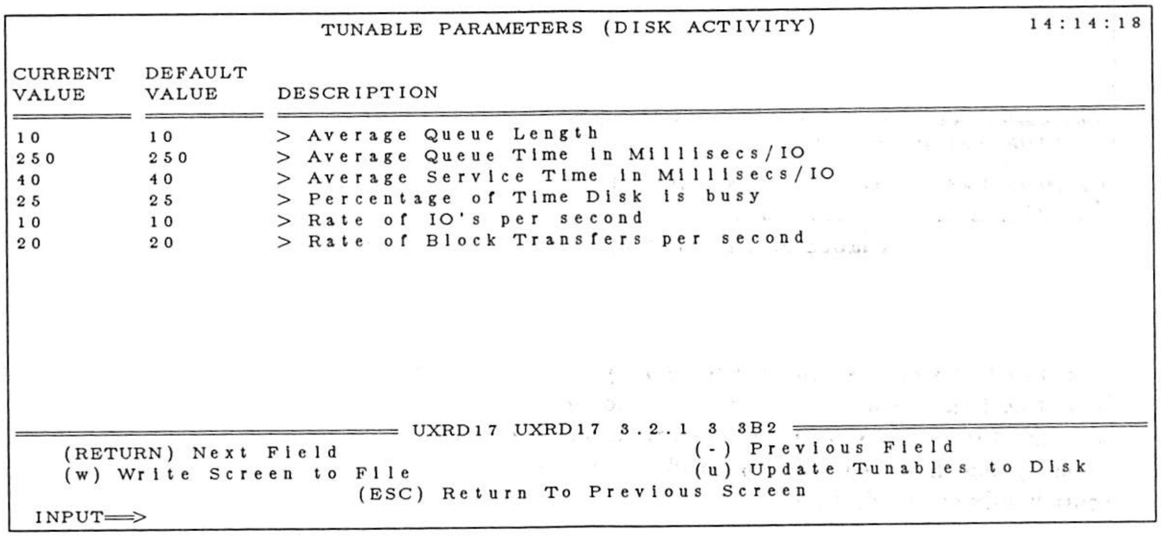
6.2 Setting Thresholds

The user may customize the intelligence facility of UFOS. When the analytic facility is invoked from any data display screen, the high- and low-water marks used to trigger highlighting can be changed for the data items specific to that screen. Some items are shared between multiple screens.

These are changed globally. After tuning the thresholds, the user has the option of disregarding these changes, using them for the current session only, or saving them to be used as the default values at subsequent invocations of UFOS.

For example, if a system is used for an application making heavy use of graphics software, the rate of character I/O may be significantly higher than the default values provided with the UFOS package. A user at this site would want to raise the rate of character I/O that will trigger highlighting, as this high rate is normal for the site and may not be cause for alarm. It is advisable for system administrator(s) to keep watch over the values encountered so that the threshold values may be updated to reflect that location's needs.

The following illustration shows the format of a tuning screen. This example corresponds to Screen 0.6.1, Disk Activity, shown in a previous illustration:



```
                         TUNABLE PARAMETERS (DISK ACTIVITY)                    14:14:18

CURRENT    DEFAULT
VALUE      VALUE      DESCRIPTION
=======    =======    =========================================================
10         10         > Average Queue Length
250        250        > Average Queue Time in Millisecs/IO
40         40         > Average Service Time in Millisecs/IO
25         25         > Percentage of Time Disk is busy
10         10         > Rate of IO's per second
20         20         > Rate of Block Transfers per second







=========================== UXRD17 UXRD17 3.2.1 3 3B2 ============================
    (RETURN) Next Field                           (-) Previous Field
    (w) Write Screen to File                      (u) Update Tunables to Disk
                          (ESC) Return To Previous Screen
  INPUT==>
```

Figure 6. Alarm Tuning Screen Format (Screen 0.6.1, Disk Activity)

6.3 Help

The *help* facility of UFOS offers explanations of data items appearing on the current screen, and what to do in many cases of unacceptable (highlighted) values. At the end of the *help* page for a data display screen, the next screen's *help* pages are presented.

7. Portability

Data collection in UFOS requires that kernel data structures be read directly from memory. Since operating system implementation varies for different UNIX System V systems, UFOS is not an easy product to port from one system to another. Many of the default thresholds used by the system must also be reevaluated.

8. Conclusion

UFOS continues the evolution of UNIX System V performance monitors, pioneered by such tools as sar, PET and PMON. These packages have varied capabilities and were developed for different purposes. UFOS was developed to satisfy the following requirements:

- Present the UNIX system in a hierarchical fashion
- Automatically direct attention to problem areas in the hierarchy
- Provide intelligent analysis of performance data with highlighting
- Provide a default base of performance thresholds
- Provide user-tunable thresholds
- Consume minimal system resources
- Provide on-line definitions of performance data values
- Aid both novice and expert users with the identification of performance problems

Due to the often complex nature of performance problems, the tool is not always effective. However, it usually does point people in the right direction. The system is intended for analyzing system-level performance problems, as opposed to application specific problems.

9. Acknowledgements

We would like to extend our thanks to Al Goddard, Nick Maniscalco, George Rette, and Eric Stewart of the AT&T IMS UNIX System Products Group for their consultations and contributions to the design of UFOS.

Also, many thanks to William Postel, Manager - AT&T IMS UNIX System Products for his continued support and encouragement.

10. References

AT&T, "Disk Performance and Tuning", UNIX System Support & Update News, September 1987.

AT&T, "Paging Parameters for the AT&T 3B2 and 3B20 Computers", UNIX System Support & Update News, May,July 1987.

AT&T, "System Tuning With SAR(1M) On the AT&T 3B2 Computer", UNIX System Support & Update News, November 1987.

"AT&T 3B2 Computer, UNIX System V Release 3, System Administrators Guide", Part 2, Chapter 6, Select Code 305-645 Issue 1, 1988.

Bach, M.J., "The Design of the UNIX Operating System", Prentice-Hall, 1986.

Chaban, E., "Using sar To Zero in on Performance Bottlenecks", UNIX WORLD, July 1988.

Farrell, B.L. and Ramamurthy, G., "A Prototype Performance Engineering / Management Tool for UNIX Based Systems", International Conference on Management and Performance Evaluation of Computer Systems, Las Vegas, 1986.

Jatkowski, P. and Akre, M., "PMON: Graphical Performance Monitoring Tool", Proceedings of the USENIX Winter Conference, Dallas, 1988.

"UNIX System V, Tuning and Configuration Guide", Select Code 307-121 Issue 1, 1984.